推酷

# 编程狂人

programming madman

NO. 31

# 关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

# 关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:

http://www.tuicool.com/mags/53b10f62d91b140354050e0c/

欢迎下载推酷客户端体验更多阅读乐趣

# 目录

# 盘点国内网站常用的一些 CDN公共库加速服务

作者:欲思

CDN公共库是指将常用的JS库存放在CDN节点，以方便广大开发者直接调用。与将JS库存放在服务器单机上相比，CDN公共库更加稳定、高速。一 般的CDN公共库都会包含全球所有最流行的开源JavaScript库，你可以在自己的网页上直接通过script标记引用这些资源。这样做不仅可以为您 节省流量，还能通过CDN加速，获得更快的访问速度。


**目前国内的一些比较大的公共CDN服务:**

## 百度CDN公共库

百度公共CDN为站长的应用程序提供稳定、可靠、高速的服务，包含全球所有最流行的开源JavaScript库。

官网：http://developer.baidu.com/wiki/index.php?title=docs/cplat/libs

Ps：百度的速度目前来说应该是和新浪差不多的。不过jQuery的版本比SAE少几个，其他类库应该都差不多。

新补充: http://yusi123.com/3125.html

## 新浪云计算CDN公共库

新浪云计算是新浪研发中心下属的部门，主要负责新浪在云计算领域的战略规划，技术研发和平台运营工作。主要产品包括 应用云平台Sina App Engine（简称SAE）。

SAE的CDN节点覆盖全国各大城市的多路（电信、联通、移动、教育）骨干网络，使开发者能够方便的使用高质量的CDN服务。

官网：http://lib.sinaapp.com/

Ps：网上有评测说新浪的速度比百度的要好。个人没感觉出来。亲测半夜的时候出现过几次加载慢的情况（不知道其他站长遇到过没有）。

## 又拍云JS库CDN服务

又拍云存储是杭州纬聚网络有限公司旗下项目，成立于2005年6月，前期主要为又拍网、又拍图片管家提供图片云存储／云计算服务，于2010年2月对所有用户开放使用。

主要专注于海量小文件的存储与分发及图片云计算领域。提供的两大核心服务：静态文件云存储、CDN加速处理。

官网：http://jscdn.upai.com/

Ps：又拍云js库提供了常用的JavaScript库CDN服务。算是起步较早的cdn加速服务了。速度和稳定性也不错。不过js库有点少，有些前卫的js库可能不提供。

## 七牛云存储 开放静态文件CDN

像 Google Ajax Library，Microsoft ASP.net CDN，SAE，Baidu，Upyun 等 CDN 上都免费提供的 JS 库的存储，但使用起来却都有些局限，因为他们只提供了部分 JS 库。但七牛云存储提供一个尽可能全面收录优秀开源库的仓库，并免费提供 CDN 加速服务。

官网：http://www.staticfile.org/

Ps：同时，开放静态文件CDN也提供开源库源接入的入口，让所有人都可以提交开源库，包括 JS、CSS、image 和 swf 等静态文件。

上面这几个，我个人测试结果:百度云应该是最快的， 又拍的服务太少，几乎不用考虑.阿里云据说也提供了公共CDN服务，但目前官网找不到具体服务页面，暂时不说了

## 360网站卫士CDN前端公共库

托管在360众多的全国CDN节点上，覆盖电信、联通、移动等主流运营商线路，您可以在自己的网页上直接通过script标记引用这些资源，让网站访问速度瞬间提速!

只需替换一个域名就可以继续使用Google提供的前端公共库和免费字体库,让网站访问速度瞬间提速。

官网：http://libs.useso.com/

Ps：360的步伐现在是越来越快了。各种技术和提供的服务更新的速度是飞快的。360CDN服务也是最近才推出的（貌似主要还是Google被墙 了的原因）。提供了大多数的前端js库，还在自己的服务器上面缓存了Google的前端公共库和免费字体库，这个算是其他国内的cdn公共库没有的。速度和稳定性也不错，当然相比较而且还是略逊于百度和新浪(亲测有些地方宽带线路会断线，可能也是很少部分吧)。毕竟提供服务还没多久。具体使用方法可查看WordPress利用360CDN公共库解决Google Open Sans字体无法加载.

目前国外的一些比较大的公共**CDN**服务：

## CDNJS

CDNJS提供非常完整的 JavaScript 程式库，无论是热门或是冷门的一应俱全。若你觉得它们缺少哪些好用的函式库，也可以自行提交到网站里，通过审核后就 CDNJS 就会为你分流 js文件！这项服务是结合 Cloud-Flare、Pingdom 与 S3Stat的，稳定性与速度自然不在话下。CDNJS提供的 JavaScript Libraries 全部列在网站首页，使用者可以直接搜索。这些程式库都有标示版本编号、标签以及原维护网站链结。

官网：http://www.cdnjs.com/

Ps：CDNJS应该算是最完整的的JS库了。存储了大部分主流的 JS 库，甚至 CSS、image 和 swf，不过很多国内优秀开源库是没有的。很多国外前卫的Js库在CDNJS大都能找到。国内的速度虽然比不上其他的几个国内的CDN服务，但是相对来说 其实还可以。

当然你也可以使用国人提供的CDNJS国内镜像网站的又拍云路径来引用相关JS和CSS文件。

国内镜像：http://www.cdnjs.cn/

CDNJS国内镜像托管在又拍云存储，但是各种JS或者CSS类库比又拍云自己出的JS库丰富很多,而且每天同步更新且支持https协议。

## Google Hosted Libraries

Google出品，必属精品了。虽然最近Google全线产品被墙了，连基本的Google搜索服务都无法使用了。但是谷歌的公共CDN公共库应该 是最强大的了，像其中的前卫的各种代码类库和Google Web Font 字体库，国内几大公共CDN服务几乎都不提供支持。

官网：https://developers.google.com/speed/libraries/

Ps：当然， Google打不开怎么办？除了使用国内的cdn库，也没有什么好办法了。如：国内cdn不提供的js库使用七牛云存储cdn加速服务、Google Fonts Open Sans字体库使用360CDN公共库代替等。

## Microsoft ASP.net CDN

ASP.NET开发团队推出的一个新的微软Ajax CDN（Content Delivery Network，内容分发网络）服务，该服务提供了对AJAX库（包括jQuery和 ASP.NET AJAX）的缓存支持。该服务是免费的，不需任何注册，可用于商业性或非商业性用途。

官网：http://www.asp.net/ajaxlibrary/cdn.ashx

Ps：微软出品，自然不会太差。虽然在天朝，速度依然不会太慢（当然比不上国内的其他cdn）。

## jsDelivr

MaxCDN是一家价格相对比较便宜的CDN公司，在全球分布着众多的节点。 jsDelivr是基于MaxCDN的一个免费开源的 CDN 解决方案，用于帮助开发者和站长。jsDelivr包含 JavaScript 库、jQuery 插件、CSS 框架、字体等等 Web 上常用的静态资源。

官网：http://www.jsdelivr.com/

Ps：每一款CDN的节点数量都是大家所关心的，jsDelivr总共提供着13个节点。加载速度和CDNJS基本差不多，国内用户建议使用国内CDN服务最佳。大家可以自己测试看看。

总结：这些CDN公共库大都各具特色。大家可以自己选择性去使用。速度和稳定性以国内的百度和新浪为最佳（当然这是个人意见）。鉴于Google已经被墙，所有关于Google的服务大家还是尽快转移阵地，使用国内的CDN公共库服务吧。

原文链接:http://yusi123.com/3093.html

# 移动端网页设计经验与心得

作者:wingkun

智能手机发展确实很迅速，像今年，我的大部分工作就都在移动端网页上。

再往前些年，看到的手机版/ 移动版网页，限制于浏览器与手机性能，2g网络速度等

网页设计无非是蓝、黑、白，界面单调，并且要尽可能的设计简单。

现在情况就大不相同了，软件上webkit内核浏览器大行其道，硬件突飞猛进，网速来说，4g正炒得火热。

下面就和大家分享一下我的一些移动端网页设计经验与心得。

## 1. 分辨率

这应该是移动端网页最关心的问题了，因为移动设备五花八门，各种分辨率都有。要想在这些设备上都能有良好的浏览体验，得花一番功夫。

- 使用view - port：这已经是移动端网页的必备了，它可以设定页面的宽度，是否允许缩放等等。

*<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=no"/>*

一般设置width=device- width，就是设置为设备的屏幕宽度，当然也可以是具体数值

- 百分比与max(min)-width使用：移动端网页不仅分辨率不一，而且随时可以横竖屏切换，所以百分比宽度设定非常必要，再配合max(min)-width限制最大(小)宽度，能有效的适应各种分辨率，若为此还有特别需求，可看下一条，"使用Media Queries"

- 使用Media Queries，这也是响应式web设计的一部分

*<link rel="stylesheet" type="text/css" href="style1.css" media="screen and (min-width: 640px)">*

这里的意思就是在大于640px的屏幕宽度下，使用style 1样式，也可以写在样式内部，如:

*@media screen and (min-width: 640px){*

*.d1{background:#ccc;}*

*}*

## 2. 内容与缓存

虽才说到现在4g正炒得火热，但不可否认移动设备网络环境上的局限性，所以还是有必要对此做一些处理。

- 使用manifest缓存 <html manifest="/mobile.manifest">

在html上添加manifest，其中文件格式内容如:

CACHE MANIFEST

##需要离线的内容

CACHE:

Script/jquery.js

Script/gameconfig.js

Image/home.png

Image/logo.png

##总是访问网络的内容

NETWORK:

*

##访问A失败时访问B

FALLBACK

浏览器将缓存chache内所有的内容，并且可以离线访问，只要文件发生任何改变都将会重新读取并刷新全部缓存，所以更改注释是个更新缓存的好方法

这里要注意的是

1，添加了manifest的当前网页也会被缓存 所以推荐的方式是页面缓存，页面动态内容全部用ajax获取，所以在移动网站项目设计开始就要注意这个问题

2，页面中添加iframe 然后子页面引用manifest想达到缓存资源而不缓存当前页面内容，是无效的。

- 尽可能使用css样式来代替图片，由于移动端浏览器对css3的支持，使得以前很多图片可以用样式来代替

如我们公司项目内用到的一些，下图：

按钮用到了 渐变+圆角+内阴影

```
.btn_red {
display: block;
line-height: 28px;
padding: 1px 0;
border: 1px solid #B81414;
border-radius: 2px;
background: #FF5A5A;
background: -webkit-gradient(linear, 0 0, 0 70%, from(#FF5A5A), to(#FF4444));
overflow: hidden;
margin-top: 3px;
color: #fff;
box-shadow: 0px 1px 1px #FFB5B5 inset;
}
```

三角形 就是用border的颜色，改变颜色可以画出指向不同方向的三角形

```
.tip_t{
border-color: transparent transparent #bb0808 transparent;
border-style: solid;
border-width: 10px;
width: 0px;
height: 0px;
}
```

箭头是两个三角形叠在一起

……

用样式代替图片之后不仅大小减少了很多，可维护性更大大提高

- 页面只展示部分内容，多提示用户"点击展开"或者"查看更多"，再异步获取内容，大家都不希望打开一个网站，流量就哗哗没了.

### 3. 布局

手机端可视区域小，布局上不同于传统网页，需要充分利用有限的空间去展示信息。

- 页面流程简单清晰，复杂的操作尽量分段展示，如下图：


- 隐藏不常用的功能，可以将其放在侧边栏或弹出层，如下图：

- 由于移动端是直接用手指操作，而非鼠标。所以，需要响应元素点击区域要相对明显，大
- 屏幕宽度虽小，但是上下滑动体验好，因此布局上可以多上下排列

## 4. 其他与结语

- 许多小厂商的手机平板，还有一些rom上的系统默认浏览器版本各异(很头疼..)，所以测试工作要做足
- html5中<input /> type有好几种新类型，比如<input type="tel" />，移动端上点击会默认弹出数字键盘，可多试试
- 本文只是总结了我项目上一些小心得，如有纰漏和错误谢谢大家指正，也欢迎大家点赞和讨论

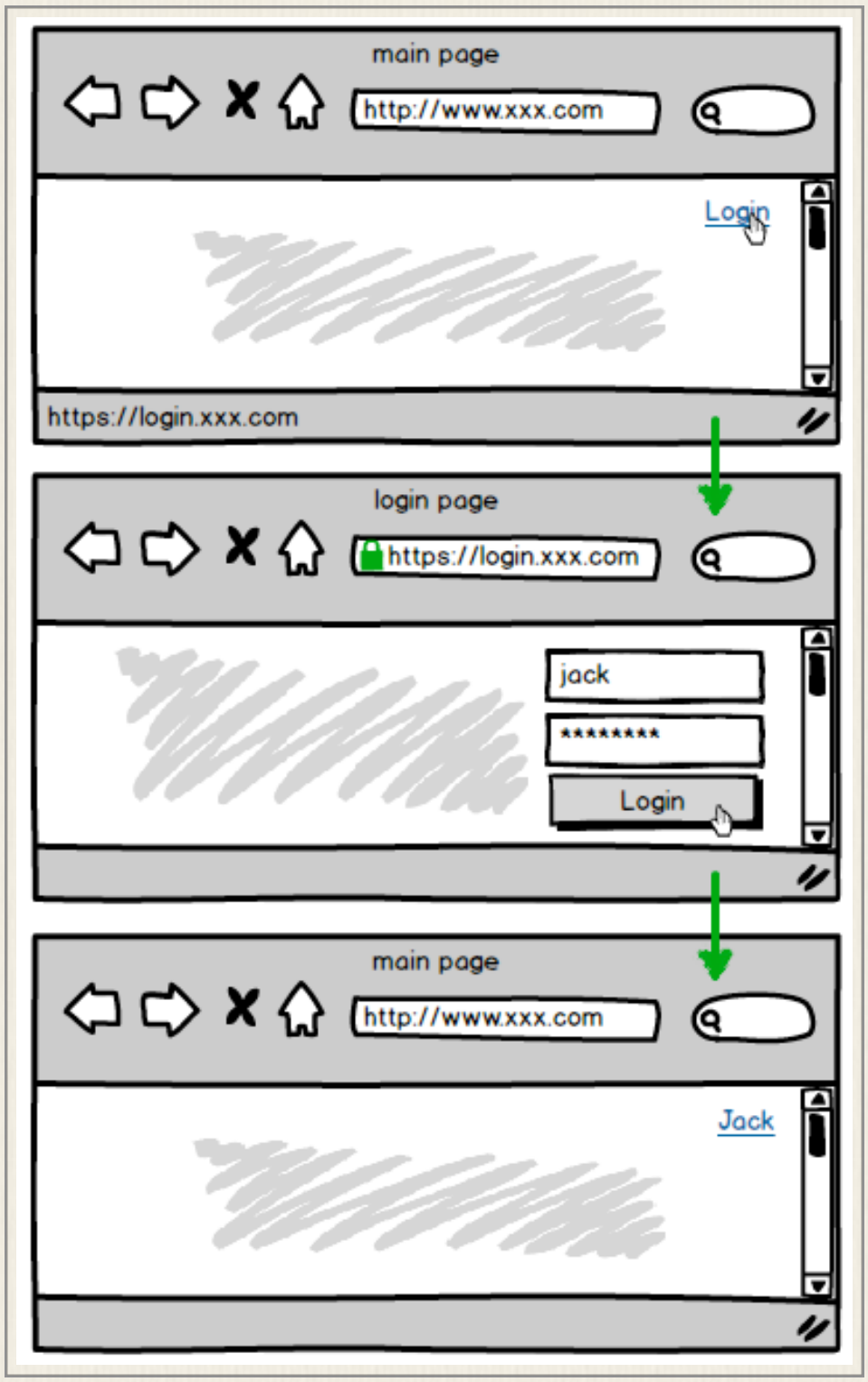
原文链接: http://www.cnblogs.com/wingkun/p/3807972.html

# 流量劫持 —— 浮层登录框的隐患

作者:zjcqoo

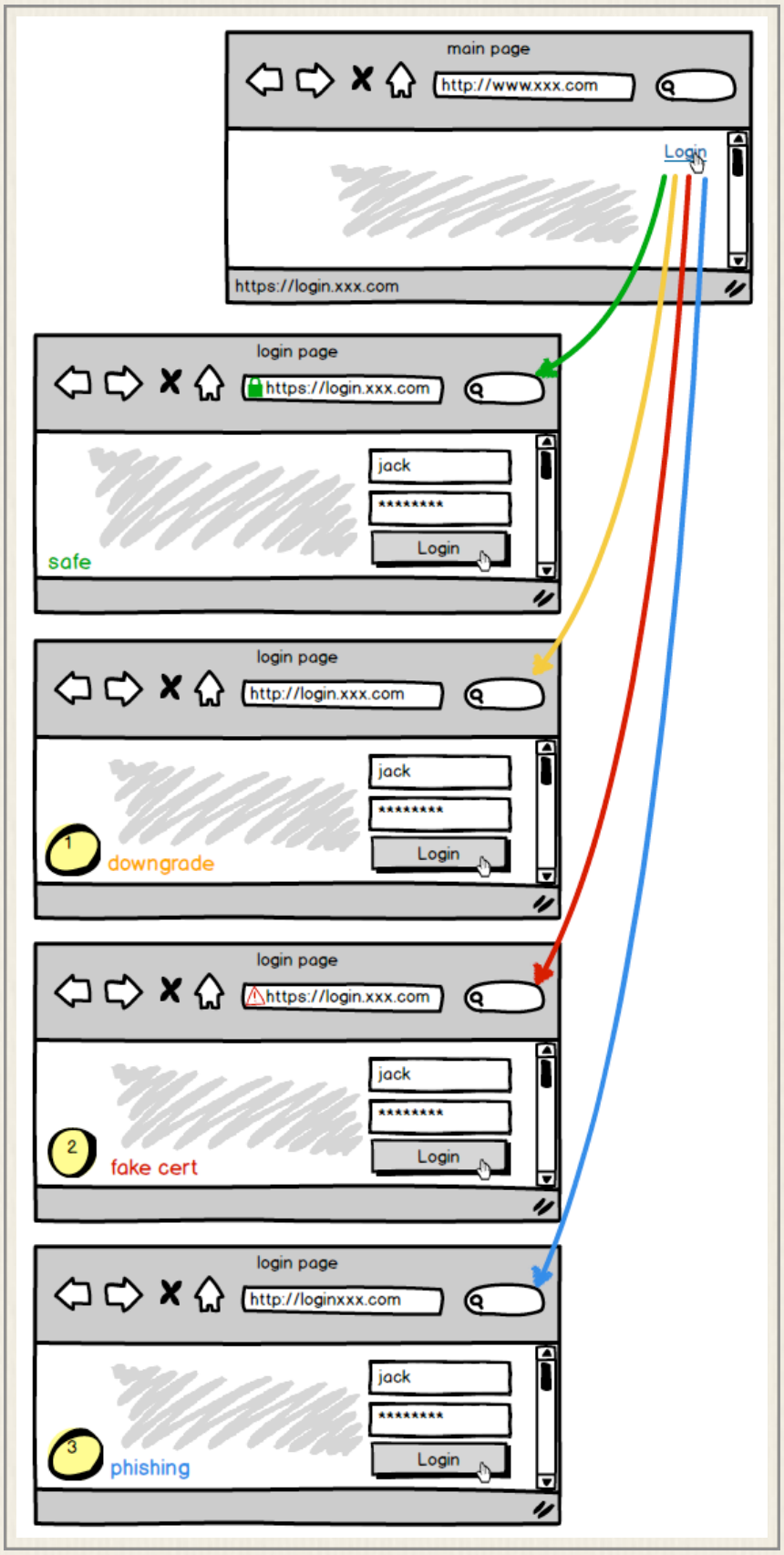
## 传统的登录框

在之前的文章流量劫持危害详细讲解了 HTTP 的高危性，以至于重要的操作都使用 HTTPS 协议，来保障流量在途中的安全。



这是最经典的登录模式。尽管主页面并没有开启 HTTPS，但登录时会跳转到一个安全页面来进行，所以整个过程仍是比较安全的 —— 至少在登录页面是安全的。

对于这种安全页面的登录模式，黑客硬要下手仍是有办法的。在之前的文章里也列举了几种最常用的方法：拦截 HTTPS 向下转型、伪造证书、跳转钓鱼网站。



其中转型 HTTPS 的手段最为先进，甚至一些安全意识较强的用户也时有疏忽。

然而，用户的意识和知识总是在不断提升的。尤其在如今各种网上交易的时代，安全常识广泛普及，用户在账号登录时会格外留心，就像过马路时那样变得小心翼翼。

久而久之，用户的火眼金睛一扫地址栏即可识别破绽。

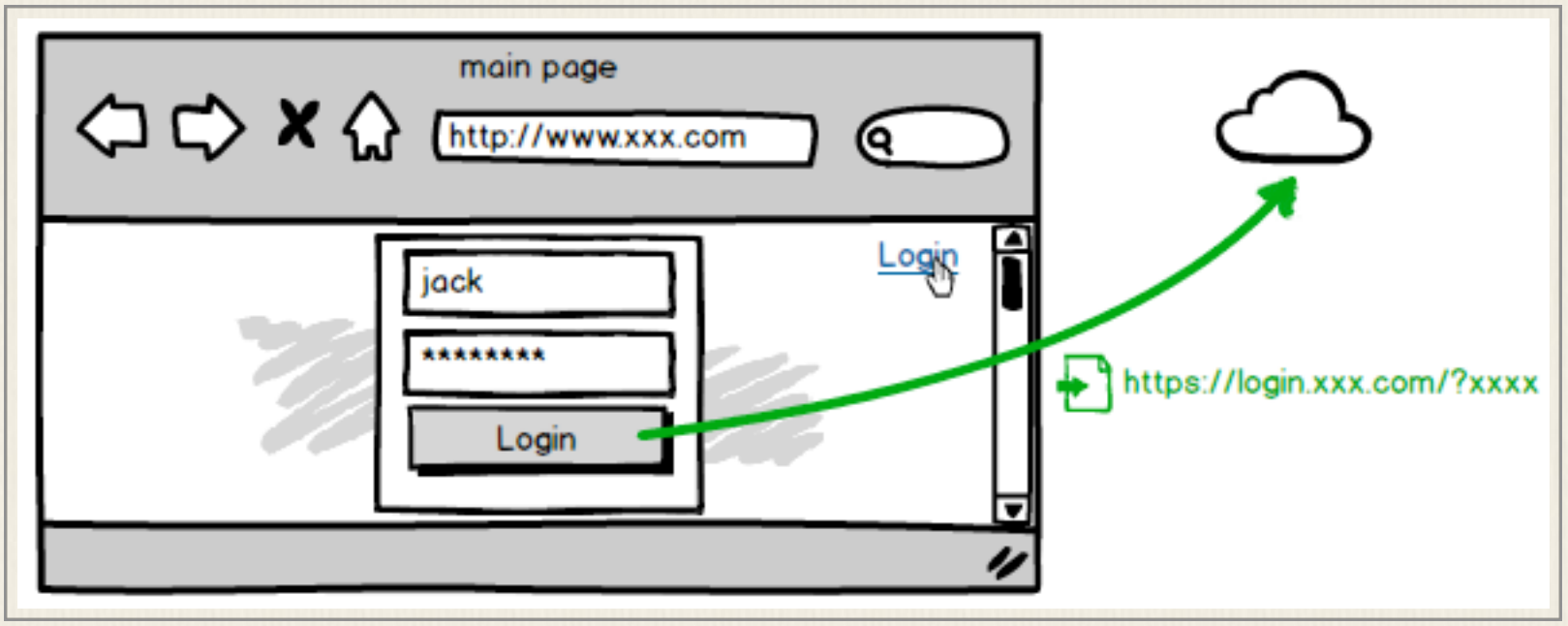

因此，这种传统的登录模式，仍具备一定的安全性，至少能给用户提供识别真假的机会。

## 华丽的登录框

不知从何时起，人们开始热衷在网页里模仿传统应用程序的界面。无论控件、窗口还是交互体验，纷纷向着本地程序靠拢，效果越做越绚。

然而华丽的背后，其本质仍是一个网页，自然掩盖不了网页的安全缺陷。

当网页特效蔓延到一些重要数据的交互 —— 例如账号登录时，风险也随之产生。因为它改变了用户的使用习惯，同时也彻底颠覆了传统的意识。




乍一看，似乎也没什么问题。虽然未使用登录页跳转，但数据仍通过 HTTPS 传输，途中还是无法被截获。

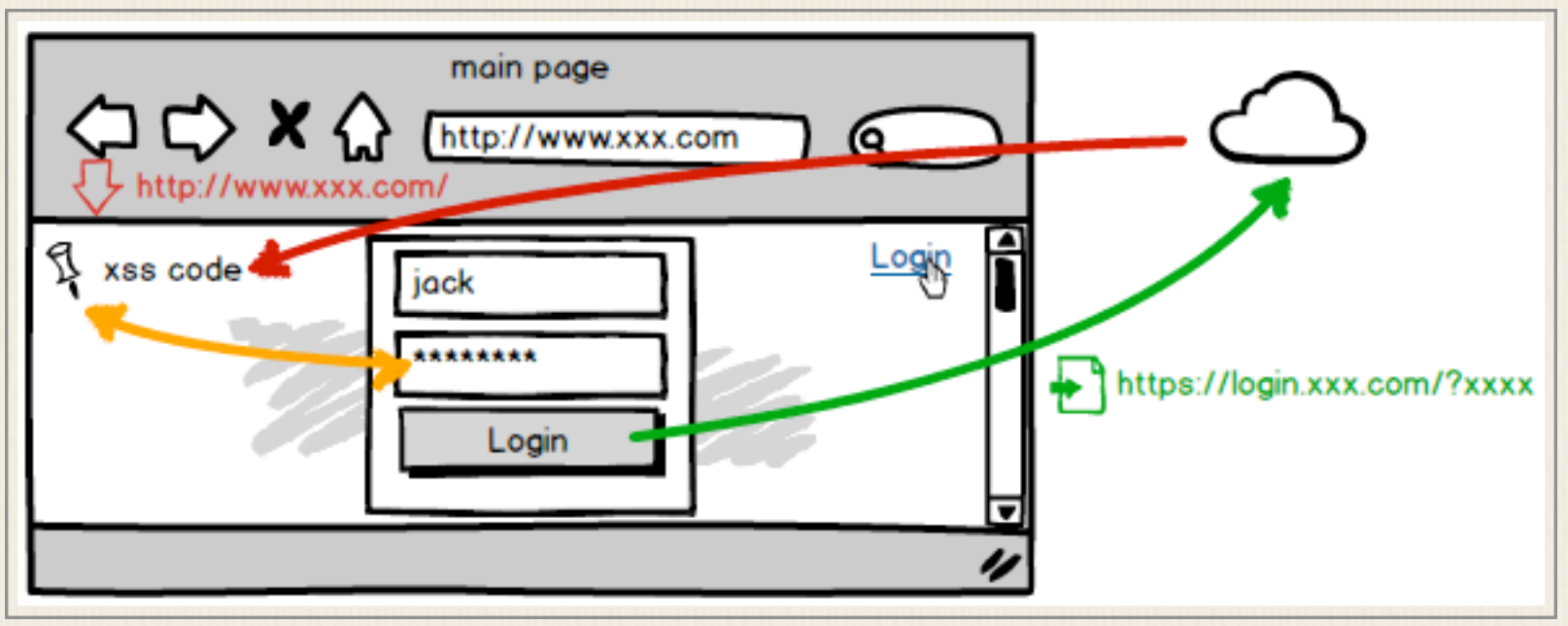
## HTTP 页面用 HTTPS 有意义吗?

如果认为这类登录框没什么大问题，显然还没领悟到『流量劫持』的精髓 —— 流量不是单向的，而是有进也有出。

能捕获你『出流量』的黑客，大多也有办法控制你的『入流量』。这在流量劫持第一篇里也详细列举了。

使用 HTTPS 确实能保障通信的安全。但在这个场合里，它只能保障『发送』的数据，对于『接收』的流量，则完全不在其保护范围内。

因为整个登录框都当作『虚拟窗口』嵌套在主页面里的，因此其中的一切都在同个页面环境里。而主页面使用的仍是不安全的 HTTP 协议，所以注入的 XSS 代码能轻而易举的控制登录框。



当然，或许你会说这只是设计缺陷。若是直接嵌入 HTTPS 登录页的 iframe 框架，那就会因同源策略而无法被 XSS 控制了。

这样的改进确实能提高一些安全性，但也只是略微的。既然我们能控制主页面，里面显示什么内容完全可以由 XSS 说了算。不论什么登录框、框架页，甚至安全插件，我们都可以将其删除，用看起来完全相同的文本框代替。得到账号后，通过后台反向代理实现登录，然后通知前 端脚本伪造一个登录成功的界面。

所以，HTTPS 被用在 HTTP 页面里，意义就大幅下降了。

## 和『缓存投毒』配合出击

在流量劫持第二篇里提到『HTTP 缓存投毒』这一概念，只要流量暂时性的被劫持，都可导致缓存长期感染。但这种攻击有个前提，必须事先找到站点下较稳定的脚本资源，做投毒的对象。

### 传统登录

在传统的登录模式里，缓存投毒非常难以利用：

HTTPS 资源显然无法被感染。

而使用 HTTPS 向下转型的方案，也会因为离开劫持环境，而无法访问中间人的 HTTP 版登陆页面，导致缓存失效；或者这个真实的 HTTP 版的登录页面根本就不接受你的本地缓存，直接重定向到正常的 HTTPS 页面。

因此只有在主页面上，修改链接地址，让用户跳转到钓鱼网站去登录，才能勉强利用。

### 浮层登录

制作一个精良的浮层登录框，需要不少的界面代码，所以经常引用 jQuery 这类通用脚本库。而这些脚本往往是长久不会修改的，因此是缓存投毒的绝好原料。

所以，浮层登录框的存在，让『缓存投毒』有了绝佳的用武之地。

在之前的文章 WiFi流量劫持 —— JS脚本缓存投毒，演示了如何利用 www.163.com 下的某个长缓存脚本进行投毒，最终利用网易的浮层登录框

获取账号。尽管网易也使用 HTTPS 传输账号数据，但在流量攻击面前不堪一击。

尽管这种登录模式风险重重，但最近百度也升级成浮层登录框，并且还是所有产品。所以，我们再次尝试那套的古老方法，看看在如今是否仍能发起攻击。

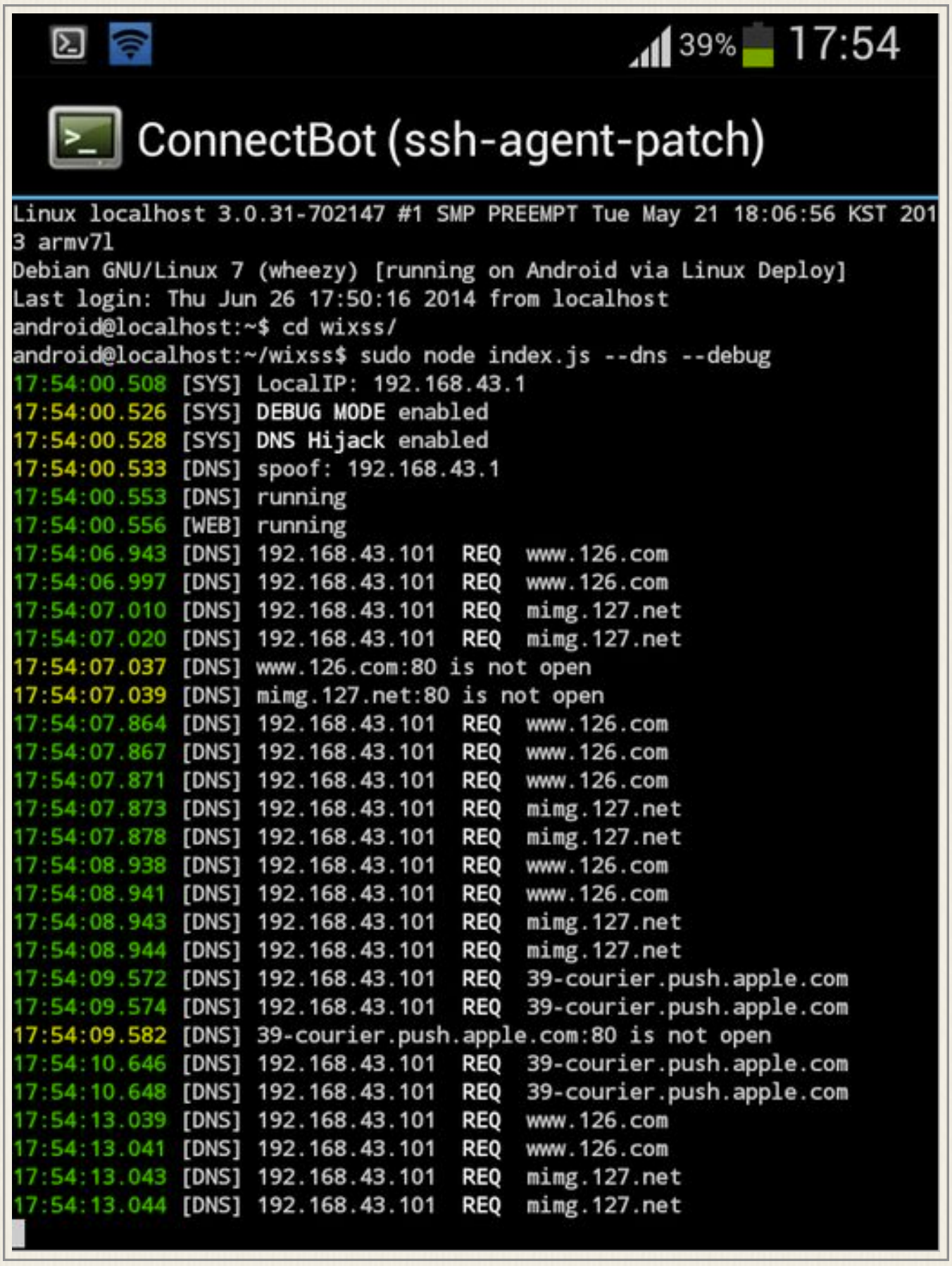
我们选几个最常用的产品线，进行一次缓存扫描：

```
D:\WebSec\wixss\tool>phantomjs sniffer.js -i baidu.txt -o ../asset/poisoning/list.json
== http://www.baidu.com ====================
   no result

== http://www.baidu.com/s?wd=ss ====================
   no result

== http://map.baidu.com ====================
   -7   / +365          http://s1.map.bdimg.com/mobile/simple/static/index/pkg/index_sync_js_0_c22
   -3   / +365          http://webmap0.map.bdimg.com/newmap/static/common/js/libs/mod_17b3ce0.js
   -3   / +365          http://webmap0.map.bdimg.com/newmap/static/common/fis_resource_map_b13f839
   -3   / +365          http://webmap0.map.bdimg.com/newmap/static/common/pkg/initmap_1da69ba.js
   -3   / +365          http://webmap0.map.bdimg.com/newmap/static/common/pkg/libs_7ed9cb7.js
   -3   / +365          http://webmap0.map.bdimg.com/newmap/static/common/pkg/init-pkg_4c86c0d.js
   -3   / +365          http://s1.map.bdimg.com/mobile/simple/static/core/pkg/core-js_c301433.js

== http://zhidao.baidu.com ====================
   -386 / +3600         http://img.baidu.com/hunter/alog/element.min.js
   -372 / +3600         http://img.baidu.com/hunter/alog/monkey.min.js
   -344 / +30           http://static.tieba.baidu.com/tb/pms/wpo.mpda.js?v=2.6
   -199 / +3600         http://img.baidu.com/hunter/m/zhidao-mobilemonkey-20131209.js
   -185 / +3600         http://img.baidu.com/hunter/m/zhidao.js?st=-16248
   -93  / +267          http://cdn.iknow.bdimg.com/static/common/lib/mod_4a8b07f.js
   -93  / +267          http://cdn.iknow.bdimg.com/static/common/pkg/framework_1d33709.js
   -91  / +269          http://cdn.iknow.bdimg.com/static/home/pkg/module_ac45e3c.js
   -57  / +303          http://cdn.iknow.bdimg.com/static/common/pkg/more_ac24367.js
   -34  / +3600         http://img.baidu.com/hunter/alog.min.js

== http://tieba.baidu.com ====================
   -344 / +30           http://static.tieba.baidu.com/tb/pms/wpo_alog_speed.js
   -322 / +30           http://tb1.bdstatic.com/tb/static-common/component/picture_rotation/pictur
   -34  / +3600         http://img.baidu.com/hunter/alog/alog.mobile.min.js
   -34  / +3600         http://img.baidu.com/hunter/alog/alog.min.js
   -33  / +3600         http://img.baidu.com/hunter/alog/dp.mobile.min.js?v=-16248
   -23  / +30           http://tb1.bdstatic.com/tb/_/interest_smiley_603ff169.js
   -17  / +3600         http://img.baidu.com/hunter/alog/dp.min.js?v=-16248
   -16  / +30           http://tb1.bdstatic.com/tb/_/image_exif_0bd01ba7.js
   -13  / +30           http://tieba.baidu.com/tb/static-common/component/commonLogic/common/cross
8.42
   -13  / +30           http://tieba.baidu.com/tb/static-common/component/commonLogic/common/user_

DONE! Found 27 results in 5 sec
```

果然，每个产品线里都有长期未修改、并且缓存很久的脚本库。

接着开启我们的钓鱼热点，让前来连接的用户，访问任何一个页面都能中毒。

为了让钓鱼热点更隐蔽，这次我们不再使用路由器，而是利用报废的安卓手机（下一篇文章详细讲解如何实现）。



为了不影响附近办公，本文就不演示同名热点钓鱼了，所以随便取了个名字。

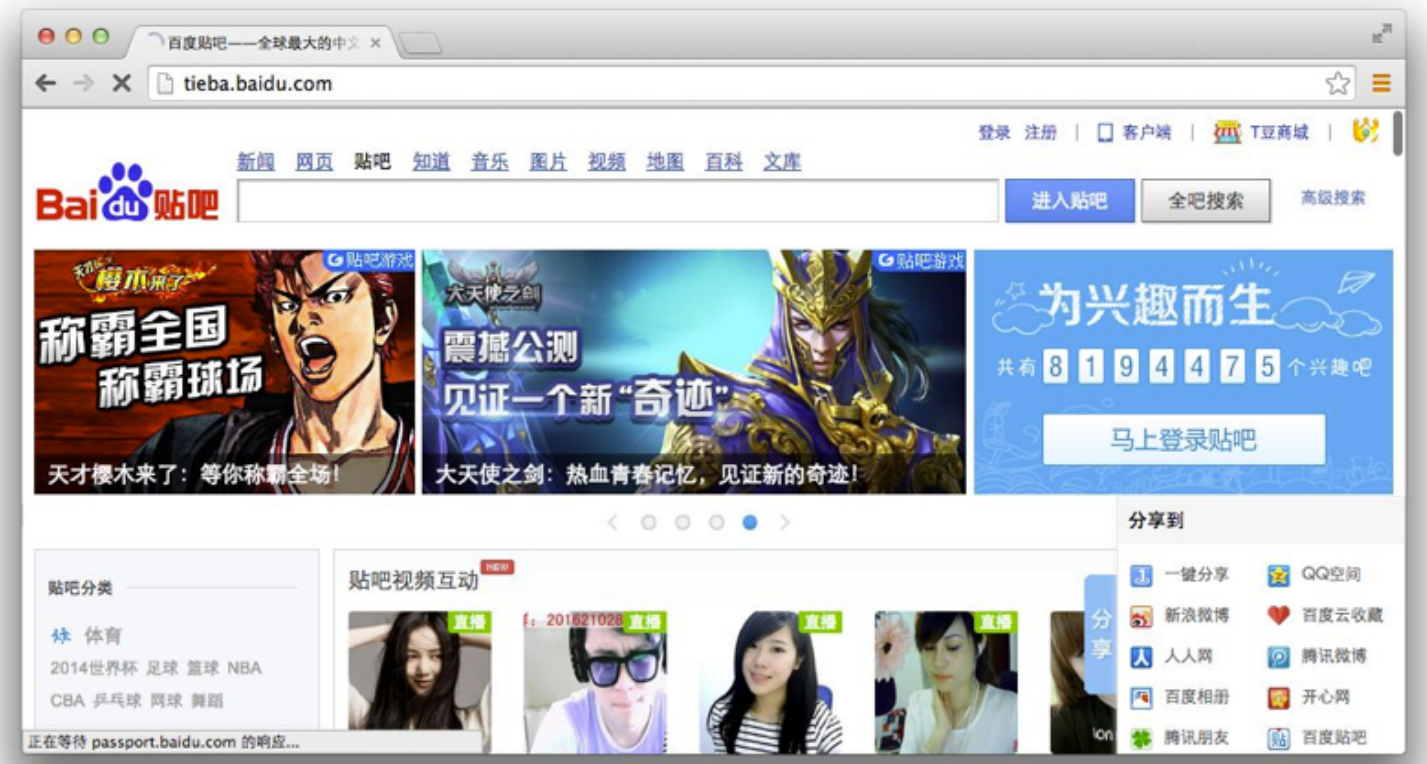
接着让『受害者』来连一下我们的热点：


之前正好开着网页，所以很快收到了 HTTP 请求。我们在任何网页里注入 XSS，进行缓存投毒。

（由于原理和之前讲一样，所以这里就省略步骤了）

然后重启电脑，连上正常的 WiFi（模拟用户回到安全的场合）。

打开 tiebai.baidu.com，一切正常。



开始登录了。。。

看看这种浮层登录框，能否躲避我们的从沉睡中唤起的 XSS 脚本：


奇迹依然发生！

由于之前有过详细的原理讲解，因此这里就不再累述了。不过在实战中，缓存投毒+非安全页面登录框，是批量获取明文账号的最理想手段。

## 不可逆的记忆

如果现在再将登录模式换回传统的，还来得及吗？显然，为时已晚。

当网站第一次从传统登录，升级到浮层登录时，用户大多不会立即输入，而是『欣赏』下这个新版本的创意。确认不是病毒广告弹出的窗口，而是真的官方设计的，才开始登录。

当用户多次使用浮层登录框之后，慢慢也就接受了这种新模式。

即使未来，网站取消了浮层登录，黑客使用 XSS 创建一个类似的浮层，用户仍会毫不犹豫的输入账号。因为在他们的记忆里，官方就曾使用过，仍然保留着对其信任度。

## 安全性升级

既然这个过程是不可逆的，撤回传统模式意义也不大。事实上，使用浮层的用户体验还是不错的，对于不了解安全性的用户来说，还是喜欢华丽的界面。

要保留体验，又得考虑安全性，最好的解决方案就是将所有的页面都使用 HTTPS，将站点武装到牙齿，不留一丝安全缝隙。这也是未来网站的趋势。

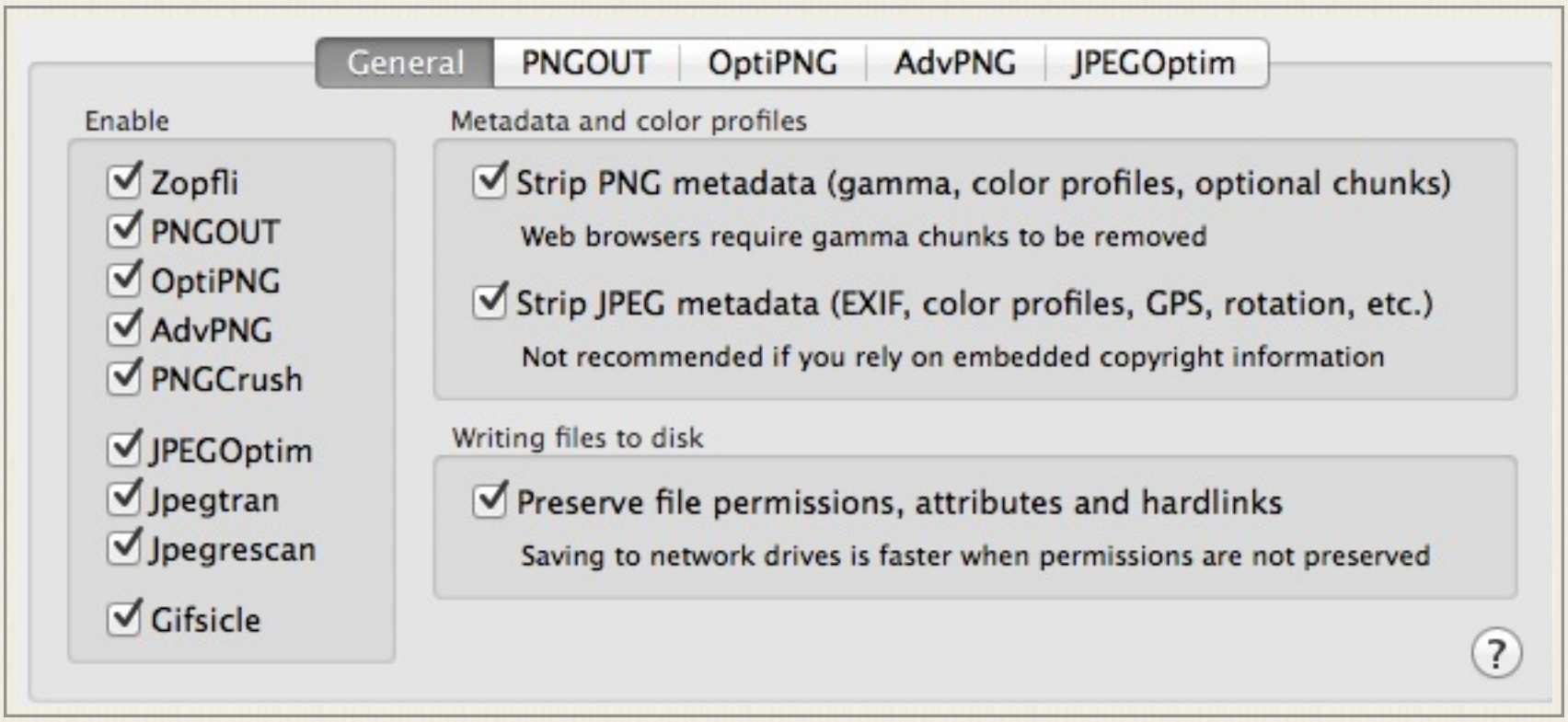
原文链接:http://fex.baidu.com/blog/2014/06/danger-behind-popup-login-dialog/

# 图片优化的那些工具

作者:晓辉

图片作为页面的一个主要因素，它的大小直接影响了页面的加载速度，这一点在移动端尤显突出。

怎么让图片的大小更小？除了选择合适的格式（jpeg、gif、png），我们还可以利用网上的应用（如smushit、tinypng、 imagemin、imageOptim）对图片进行压缩。在这里我想给大家介绍一下上述应用主要使用到了哪些命




令行工具以及它们的使用方法。

## Jpegtran

JPEG的压缩工具有jpegtran和jpegoptim，这两款工具的压缩效果几乎没有区别，在这里我们推荐使用jpegtran，相比后者，jpegtran可以进行progressive编码，使图片渐进式的展现，先显示模糊的图片，再逐步清晰。

推荐命令行参数：

jpegtran –copy none –optimize -progressive -outfile out.jpg in.jpg

想知道这些参数的具体作用，可使用命令“jpegtran –h”了解，下同。

## Gifsicle

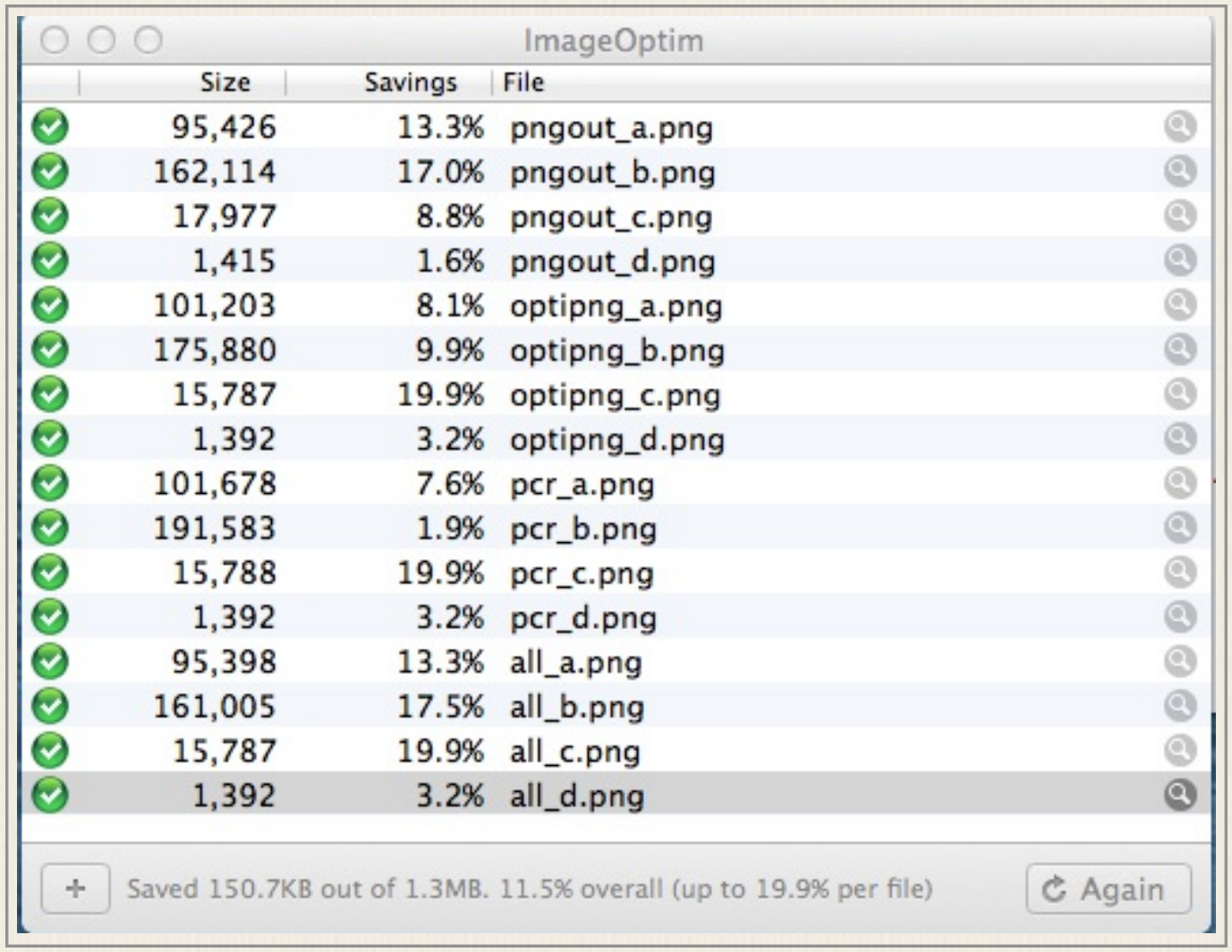
Gif动画可使用gifsicle来优化，它会剥离不同帧中重复的像素来优化gif动画，对于单帧gif我们推荐还是使用png8来替代。

推荐命令行参数：

gifsicle –interlace -O3 –careful –no-comments –no-names –same-delay –same-loopcount –no-warnings -o out.gif in.gif

## pngcrush、optipng、pngout

PNG压缩可分为无损压缩和有损压缩，以上三款可以说是现在比较主流的无损压缩工具了。从ImageOptim的压缩效果可以看出，optipng 和pngcrush对于色彩比较单一、大小比较小的图片压缩效果好于pngout，而对于色彩比较丰富、透明渐变的图片来说pngout的压缩比明显更 高。另外，经测试，Google的PageSpeed上提供的可压缩比是按照optipng给出的。




推荐命令行参数:

pngcrush -brute -rem alla -nofilecheck -bail -blacken -reduce -cc in.png out.png

optipng -strip all -quiet -clobber -o3 -i0 in.png -out out.png

pngout -k1 -r -v in.png out.png

## pngquant、pngnq

两款PNG的有损压缩工具，基本都能将图片压缩掉40%以上，它们会将PNG转换成alpha透明的PNG8，由于PNG8最多支持256色，所以 内容丰富的图片压缩后会看出些许差异，但属于可接受范围内，而纯色图片基本能保持原图的质量。另外，这种alpha透明的PNG8图片在IE6以上及其他 标准浏览器可以显示正常的alpha透明度，在IE6中则会忽略掉有alpha透明度的颜

色，作为全透明处理（边缘稍有锯齿但影响不大），而不像 png32那样alpha透明区域在IE6下显示灰色。

推荐命令行参数：

pngnq –s 1 –d outdir/ in.png

pngquant -s1 –o out.png in.png

PS：pngquant的-s是speed参数，可选值1-10，默认为3，在经过多图测试后发现1的压缩比和效果都是最佳的，其他的参数或多或少存在缺陷，这里推荐选1。

## 总结

如果您已经学会如何使用这些命令行软件对自己的图片进行压缩优化，那么恭喜您，您的图片瘦身成功。如果您觉得命令行一行一行的压缩图片太麻烦，那么 除了去使用文章开头所说到的那几款应用外，感兴趣的同学也可以整合它们去开发一套自己的应用，利用php的exec、nodejs的child_process.exec(cmd, [options], callback)等等方法执行shell命令，再配上一些交互，一款好用的图片优化应用就此诞生。最后希望这篇文章对大家有所帮助。

原文链接:http://ued.ctrip.com/blog/?p=3582

# 浅析Java 8的聚合操作

作者:赵永

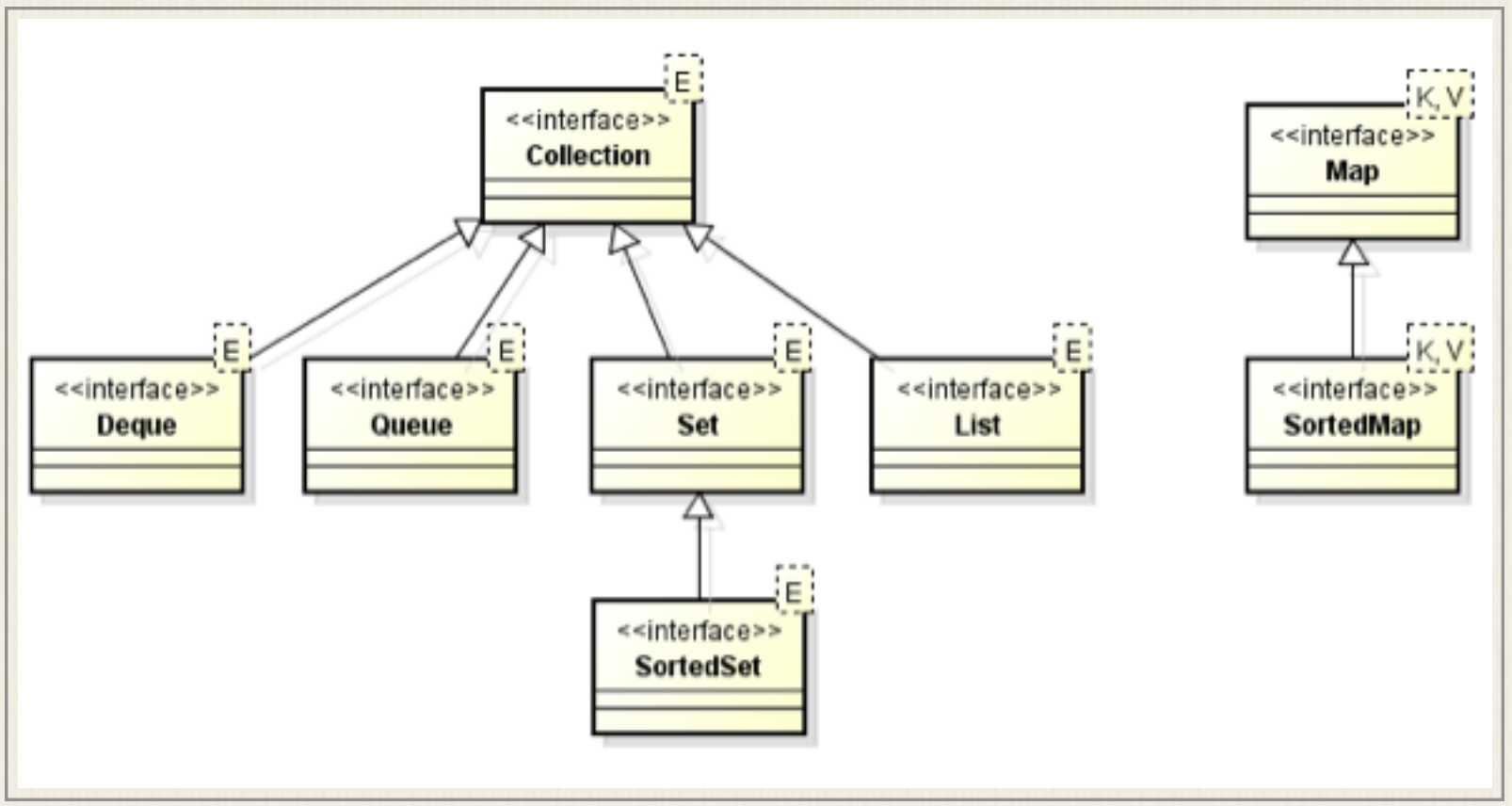
Oracle在2014年3月19日如期发布了Java 8。Java 8版本被认为是具有里程碑意义的一个版本，Oracle在该版本中添加了许多新特性，包括Lambda表达式、方法引用、加强了安全等等。

在众多的新特性中，聚合操作（Aggregate Operations）是针对集合类的一个比较大的变化。通过聚合操作，开发者可以更容易地使用Lambda表达式，并且更方便地实现对集合的查找、遍历、过滤以及常见计算等。

聚合操作与Java 8中的Lambda表达式、方法引用等新特性是相关的，一般一起组合使用，但这里只说明聚合操作的使用，下面就聚合操作的使用进行简单说明。

## 集合类的层次结构

集合类是Java语言提供的辅助类，是一种较为通用的数据结构，如Map、Set、List等。Java中集合类层次关系如下:



如上图，Collection是主要集合类的接口，其子接口（具化接口）有Deque、Queue、Set、List等。

Map是另一种类型的集合，以Key、Value的键值对存储数据集。

在Java 8中，在java.util.Collection接口中添加了如下方法：

```
Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

stream()方法的可见性修饰符为default，这又是Java 8的新特性。在接口中（Collection为interface），本不需要（也不能）进行方法实现，但引入default修饰后就不同了。开发者不但 可以进行方法的实现，而且还不用考虑向后兼容的问题。关于Default Method的详细解释，读者可以参考Java 8的官方文档。

正是stream方法引出了集合类的聚合操作。

[注意]

Map接口中并没有stream()方法，但是Map的values()和keySet()均返回集合对象，在集合对象上当然是可以使用stream()方法的。

## 聚合操作实例

为说明聚合操作的使用，首先定义一个数据元素类Person，如下：

```
import java.time.LocalDate;


public class Person {
        String name;
        LocalDate birthday;
        Sex gender;
        String emailAddress;
```

```
    public int getAge() {
        return LocalDate.now().getYear() - birthday.getYear();
    }

    public void setBirthday(LocalDate birthday){
        this.birthday = birthday;
    }

    public void setGender(Sex sex){
        this.gender = sex;
    }

    public void printPerson() {
        System.out.println("The name is " + name);
    }

    public Sex getGender(){
        return gender;
    }

    public enum Sex {
        MALE, FEMALE
    }
}
```

在Java 8以前的版本中，对Person集合的遍历往往采用以下方式：

```
Set<Person> persons = new HashSet<Person>();
//传统遍历方式 for (Person person : persons) {    if (person.getAge() > 18) {          System.out.println(person.name + " is elder than 18.");     } }
```

同样的功能，在Java 8中使用聚合操作，可以实现如下：

```
//使用聚合操作
persons.stream().filter(new Predicate<Person>() {
    @Override
        public boolean test(Person person) {
            if (person.getAge() > 18) {
                return true;
            } else {
                return false;
            }
        }
    }).forEach(new Consumer<Person>() {
        @Override
        public void accept(Person person) {
            System.out.println(person.name + " is elder than 18.");
        }
    });
```

首先，在集合对象persons上调用stream()方法（聚合操作），取得person对象的数据集（elements），然后调用聚合操作filter()对集合中的元素进行过滤，再调用forEach()完成对符合条件的person的打印。

Predicate和Consumer为Java 8中定义的函数接口(Functional Interface)，在java.util.function包下面，函数接口也是Java 8的新特性。在上述代码中，使用了两个匿名类分别对Predicate和Consumer进行了实现，这两个接口都只有一个方法，这也是函数接口的特征之 一。

上述代码中的写法还是比较繁琐的，为进一步简化，可以使用Lambda表达式实现，如下：

```
// 使用聚合操作及Lambda
  persons.stream()
      .filter(p -> p.getAge() >= 18)
      .forEach(p -> System.out.println(p.name + " is elder than 18."));
```

因为filter()、forEach()的参数均为函数接口，所以可以替换为Lambda表达式的方式。简单来理解，Lambda表达式就是允许开发者将代码逻辑作为参数进行传递，关于Lambda表达式的详细内容，请参Java 8的官方文档。

## 聚合操作的使用

聚合操作是Java 8针对集合类，使编程更为便利的方式，可以与Lambda表达式一起使用，达到更加简洁的目的。

前面例子中，对聚合操作的使用可以归结为3个部分：

1. 数据源部分：通过stream()方法，取得集合对象的数据集。

2. 通过一系列中间（Intermediate）方法，对数据集进行过滤、检索等数据集的再次处理。如上例中，使用filter()方法来对数据集进行过滤。

3. 通过最终（terminal）方法完成对数据集中元素的处理。如上例中，使用forEach()完成对过滤后元素的打印。

中间方法除了filter()外，还有distinct()、sorted()、map()等等，其一般是对数据集的整理（过滤、排序、匹配、抽取等等），返回值一般也是数据集。

最终方法往往是完成对数据集中数据的处理，如forEach()，还有allMatch()、anyMatch()、findAny()、 findFirst()，数值计算类的方法有sum、max、min、average等等。最终方法也可以是对集合的处理，如

reduce()、 collect()等等。reduce()方法的处理方式一般是每次都产生新的数据集，而collect()方法是在原数据集的基础上进行更新，过程中不 产生新的数据集。

从上面的例子中可以看出，通过stream()方法，从集合对象获取的数据集与集合对象的迭代器（Iterator）有些类似，但他们也不完全相同：

1. 迭代器提供next()、hasNext()等方法，开发者可以自行控制对元素的处理，以及处理方式，但是只能顺序处理；

2. stream()方法返回的数据集无next()等方法，开发者无法控制对元素的迭代，迭代方式是系统内部实现的，同时系统内的迭代也不一定是顺序的，还可以并行，如parallelStream()方法。并行的方式在一些情况下，可以大幅提升处理的效率。

除上述介绍的聚合操作外，Java 8中还提供了其他更为丰富的聚合操作，读者可以参考Java 8的开发参考，了解更多内容。

## 总结

Java 8提供的聚合操作，以及一起使用的Lambda表达式为开发者带来了便利，尤其在面向逻辑易变、开发迭代较快的项目应用时。但笔者个人认为，在带来方便的 同时，可能也带来了一些麻烦，如相同逻辑的复用，以及代码的查错、修改等，当然这些问题也是相对而言的。毕竟，任何事物都有两面性，技术在不断的发 展，Java也在不断地调整自己的适应性，变得功能越来越多，越来越强大了。

感谢张龙对本文的审校。

原文链接:http://www.infoq.com/cn/articles/Java8-aggregation-operation

# GCC优化引起的一个”问题”

作者:Laruence

一切要从今天下午5点左右说起, 调试一个扩展, 用valgrind (valgrind-3.8.1)做例行检查, 很不幸的valgrind报告invalid read:

```
==6467== Invalid read of size 8
==6467==    at 0x82F3B5: ZEND_JMPZ_SPEC_VAR_HANDLER (zend_execute.h:99)
==6467==    by 0x81A117: execute_ex (zend_vm_execute.h:358)
==6467==    by 0x7EA784: zend_execute_scripts (zend.c:1314)
==6467==    by 0x784D78: php_execute_script (main.c:2550)
==6467==    by 0x8870E5: do_cli (php_cli.c:980)
==6467==    by 0x88778C: main (php_cli.c:1359)
==6467==  Address 0x5d5d300 is 16 bytes inside a block of size 22 alloc'd
==6467==    at 0x4A078B8: malloc (vg_replace_malloc.c:270)
```

db attach上去以后, 发现报告错误的地方是:

```
        case IS_STRING:
            if (Z_STRLEN_P(op) == 0
                || (Z_STRLEN_P(op)==1 && Z_STRVAL_P(op)[0]=='0')) {
                result = 0;
            } else {
                result = 1;
            }
```

因为在PHP NG(PHP New Generation)中, 使用了新的字符串结构来保存字符串, 也就是zend_string:

```
struct _zend_string {
    zend_refcounted   gc;
    zend_ulong        h;                /* hash value */
    int               len;
    char              val[1];
};
```

而排查了半天, 我确认这个op是经过正常初始化的, 那问题出在哪里呢?

突然看到op是一个长度为1的字符串”0″, 就突然想起来, 之前我们做了个很”精细”的优化, 因为对于上面的结构体, 在64位的系统上, sizeof它, 由于padding, 实际上会得到大于8 + 8 + 4 + 1(21) 的大小(8 + 8 + 8 = 24).

所以我们不会使用一般来说的做法:

```
str = malloc(sizeof(str) + len + 1)
```

来为一个长度为len的字符串申请内存. 而是会使用类似:

```
str = malloc ((int)((str*)0)->val) + len + 1)
```

的方式来为一个字符串申请内存, 所以对于”0″, 我们实际上申请分配的内存是22bytes.

但, 又会有什么问题呢? 于是让我们再次db attach上去, disassmble 下看看具体是什么原因:

```
0x000000000082f3ab <ZEND_JMPZ_SPEC_VAR_HANDLER+347>:    mov    $0xffffffffff,%rax
0x000000000082f3b5 <ZEND_JMPZ_SPEC_VAR_HANDLER+357>:    and    0x10(%rdx),%rax
0x000000000082f3b9 <ZEND_JMPZ_SPEC_VAR_HANDLER+361>:    mov    $0x3000000001,%rdx
0x000000000082f3c3 <ZEND_JMPZ_SPEC_VAR_HANDLER+371>:    cmp    %rdx,%rax
```

恩, 问题就出在f3b5这行, GCC读取了0×10(%rdx)位置上的一个word大小的数据, %rdx此时是zend_string op的指针, 而0×10偏移是str->len. 原来是因为GCC优化很聪明的把

```
if (str->len == 1 && str->val[0] == '0')
```

优化成了和一个数据0×3000000001比较的一条指令….

于是, 如上面所说, 因为这个str只有22个bytes, 当尝试从16偏移处尝试读取8个字节的时候, 我们其实多读了str结构体外面的3个字节…… 于是就invalid read了

问题清楚了, GCC聪明的优化, 引起的一个无害的报告(and 0xffffffffff) …… 于是, 白忙活了…. (当然, 最好还是修复掉, 我现在打算的修复就是, 最小也要分配一个24bytes).

原文链接: http://www.laruence.com/2014/06/26/2955.html

# 分布式存储系统的雪崩效应

作者:马冠南

## 一 分布式存储系统背景

副本是分布式存储系统中的常见概念：将一定大小的数据按照一定的冗余策略存储，以保障系统在局部故障情况下的可用性。

副本间的冗余复制方式有多种，比较常用有两类：

- Pipeline：像个管道，a->b->c，通过管道的方式进行数据的复制。该方式吞吐较高，但有慢节点问题，某一节点出现拥塞，整个过程都会受影响
- 分发：client -> a  client ->b client ->c。系统整体吞吐较低，但无慢节点问题

对于冗余副本数目，本文选择常见的三副本方案。

分布式存储系统一般拥有自动恢复副本的功能，在局部存储节点出错时，其他节点（数据副本的主控节点或者client节点，依副本复制协议而定）自动 发起副本修复，将该宕机存储节点上的数据副本恢复到其他健康节点上。在少量宕机情况下，集群的副本自动修复策略会正常运行。但依照大规模存储服务运维经 验，月百分之X的磁盘故障率和月千分之X的交换机故障率有很大的可能性导致一年当中出现几次机器数目较多的宕机。另外，批量升级过程中若出现了升级 bug，集群按照宕机处理需要进行副本修复，导致原本正常时间内可以完成的升级时间延长，也容易出现数目较多的宕机事件。

## 二 雪崩效应的产生

在一段时间内数目较多的宕机事件有较大可能性诱发系统的大规模副本补全策略。目前的分布式存储系统的两个特点导致这个大规模副本补全策略容易让系统产生雪崩效应：

a. 集群整体的free空间较小：通常整体<=30%, 局部机器小于<=20%甚至10%

b. 应用混布：不同的应用部署在同一台物理/虚拟机器上以最大化利用硬件资源

今年火起来的各种网盘、云盘类服务就是a的典型情况。在各大公司拼个人存储容量到1T的背后，其实也在拼运营成本、运维成本。现有的云存储大多只增 不减、或者根据数据冷热程度做数据分级（类似Facebook的数据分级项目）。云存储总量大，但增量相对小，为了减少存储资源和带宽资源浪费，新创建的 文件若原有的存储数据中已有相同的md5或者sha1签名则当做已有文件做内部链接，不再进行新文件的创建。但即使这样，整体的数据量还是很大。

目前云存储相关业务未有明显的收入来源，每年却有数万每台的服务器成本，为运营成本的考虑，后端分布式存储系统的空闲率很低。而瞬间的批量宕机会带 来大量的副本修复，大量的副本修复很有可能继而打满原本就接近存储quota的其他存活机器，继而让该机器处于宕机或者只读状态。如此继续，整个集群可能 雪崩，系统残废。

## 三 预防雪崩

本节主要讨论如何在系统内部的逻辑处理上防止系统整体雪崩的发生。预防的重要性大于事故之后的处理，预测集群状态、提前进行优化也成为预防雪崩的一个方向。

下面选取曾经发生过的几个实际场景与大家分享。

### 1. 跨机架副本选择算法和机器资源、用户逻辑隔离

**现场还原：**

某天运维同学发现某集群几十台机器瞬间失联，负责触发修复副本的主控节点开始进行疯狂的副本修复。大量用户开始反馈集群变慢，读写夯住。

**现场应对:**

优先解决——副本修复量过大造成的集群整体受影响。

a. 处理的工程师当机立断，gdb到进程更改修复副本的条件为副本<2，而非原本的3（replicas_num），让主控节点这个时候仅修复副本数小 于2个的文件，即保证未丢失的文件有至少一个冗余副本，防止只有一个副本的数据因可能再次发生的挂机造成文件丢失。

b. 紧急解决这批机器失联问题，发现是交换机问题，a.b.c.d ip网段的c网段机器批量故障。催促网络组尽快修复。

c. 副本修复到>=2之后，Gdb更改检测副本不足周期，将几十秒的检测时间推迟到1天。等待网络组解决交换机问题。

d. 网络恢复，原有的机器重新加入集群。大量2副本文件重新变为3副本，部分3副本全丢失文件找回。

e. 恢复主控节点到正常参数设置状态，系统开始正常修复。

**改进措施:**

在改进措施前，先分析下这次事件暴露的系统不足:

1) Master参数不支持热修正，Gdb线上进程风险过大。

2) 一定数量但局域性的机器故障影响了整体集群（几十台相对一个大集群仍属于局域性故障）。如上所述，月千分之几的故障率总有机会让你的存储系统经历一次交换机故障带来的集群影响。

案例分析后的改进措施出炉:

1) Master支持热修正功能排期提前，尽早支持核心参数的热修改。

热修改在上线后的效果可观，后续规避过数次线上问题。

2) 在选择数据副本存储宿主机器的pickup算法中加入跨交换机（机架位）策略，强制——或者尽量保证——副本选择时跨机架位。这种算法底下的副本，至少有 1个副本与其他两个副本处于不同的交换机下（IP a.b.c.d的

c段）。该措施同时作用于新的存储数据副本选择和副本缺失后的副本补全策略，能在副本宿主选择上保证系统不会因为交换机的宕机而出现数据 丢失，进而避免一直处于副本补全队列/
列表的大量的丢失副本节点加重主控节点负载。

3) 机器按region划分隔离功能提上日程；用户存储位置按照region进行逻辑划分功能提上日程；Pickup算法加入跨region提上日程。

a) 机器按照物理位置划分region、用户按照region进行逻辑存储位置划分，能让集群在局部故障的情况下仅影响被逻辑划分进使用这部分机器的用户。

这样一来，最坏情况无非是这个region不可用，导致拥有这个region读写权限的用户受影响。Pickup算法跨region的设计进一步保 证被划分region的用户不会因为一个region不可用而出现数据丢失，因为其他副本存到其他region上了。于是，核心交换机故障导致一个 re-
gion数百台机器的宕机也不会对集群造成范围过大的影响了。

b) 增加region 可信度概念，将机器的稳定性因素加入到副本冗余算法中。

当集群规模达到一定量后，会出现机器稳定性不同的问题（一般来说，同一批上线的机器稳定性一致）。通过标记region的稳定性，能强制在选择数据副本的时候将至少一个副本至于稳定副本中，减少全部副本丢失的概率。

c) Region划分需要综合考虑用户操作响应时间S-
LA、物理机器稳定情况、地理位置等信息。

合理的region划分对提升系统稳定性、提升操作相应时间、预防系统崩溃都有益处。精巧的划分规则会带来整体的稳定性提升，但也增加了系统的复杂度。这块如何取舍，留给读者朋友深入思考了。

### 2. 让集群流控起来

流控方面有个通用且符合分布式存储系统特点的原则：任何操作都不应占用过多的处理时间。这里的“任何操作”包含了在系统出现流量激增、局部

达到一定数量的机器宕机时进行的操作。只有平滑且成功的处理这些操作，才能保证系统不因为异常而出现整体受影响，甚至雪崩。

**现场还原：**

1) 场景1 某天运维同学发现，集群写操作在某段时间大增。通过观察某个存储节点，发现不仅是写、而且是随机写！某些产品线的整体吞吐下降了。

2) 场景2 某集群存储大户需要进行业务调整，原有的数据做变更，大量数据需要删除。

运维同学发现，a. 整个集群整体上处于疯狂gc垃圾回收阶段 b. 集群响应速度明显变慢，特别是涉及到meta元信息更新的操作。

3) 场景3 某天运维同学突然发现集群并发量激增，单一用户xyz进行了大量的并发操作，按照原有的用户调研，该用户不应该拥有如此规模的使用场景。

此类集群某些操作预期外的激增还有很多，不再累述。

**现场应对：**

1) 立刻电联相关用户，了解操作激增原因，不合理的激增需要立刻处理。

我们发现过如下不合理的激增：

a. 场景1类：通过Review代码发现，大量的操作进行了随机读写更改。建议用户将随机读写转换为读取后更改+写新文件+删除旧文件，转换随机读写为顺序读写。

b. 场景3类：某产品线在线上进行了性能测试。运维同学立刻通知该产品线停止了相关操作。所有公有集群再次发通过邮件强调，不可用于性能测试。如有需要，联系相关人员在独占集群进行性能场景测试。

2) 推动设计和实现集群各个环节的流控机制功能并上线。

**改进措施：**

1) 用户操作流控

a. 对用户操作进行流控限制

可通过系统内部设计实现，也可通过外部的网络限流等方式实现，对单用户做一定的流控限制，防止单个用户占用过多整个集群的资源。

b. 存储节点操作流控

可按照对集群的资源消耗高低分为High – Medium – Low三层，每层实现类似于抢token的设计，每层token数目在集群实践后调整为比较适合的值。这样能防止某类操作过多消耗集群负载。若某类操作过 多消耗负载，其他操作类的请求有较大delay可能，继而引发timeout后的重试、小范围的崩溃，有一定几率蔓延到整个集群并产生整体崩溃。

c. 垃圾回收gc单独做流控处理。删除操作在分布式存储系统里面常用设计是：接收到用户删除操作时，标记删除内容的meta信息，直接回返，后续进行策略控 制，限流的删除，防止大量的gc操作消耗过多单机存储节点的磁盘处理能力。具体的限流策略和token值设置需要根据集群特点进行实践并得出较优设置。

2) 流控黑名单

用户因为对线上做测试类的场景可以通过人为制度约束，但无法避免线上用户bug导致效果等同于线上测试规模的场景。这类的场景一般在短时间内操作数严重超过限流上限。

对此类场景可进行流控黑名单设置，当某用户短时间内（e.g. 1小时）严重超过设置的上限时，将该用户加入黑名单，暂时阻塞操作。外围的监控会通知运维组同学紧急处理。

3) 存储节点并发修复、创建副本流控

大量的数据副本修复操作或者副本创建操作如果不加以速度限制，将占用存储节点的带宽和CPU、内存等资源，影响正常的读写服务，出现大量的延迟。而大量的延迟可能引发重试，加重集群的繁忙程度。

同一个数据宿主进程需要限制并发副本修复、副本创建的个数，这样对入口带宽的占用不会过大，进程也不会因为过量进行这类操作而增加大量其他操作的延 迟时间。这对于采用分发的副本复制协议的系统尤其重要。分发协议一般都有慢节点检查机制，副本流控不会进一步加重系统延迟而增大

成为慢节点的可能。如果慢 节点可能性增大，新创建的文件可能在创建时就因为慢节点检查机制而缺少副本，这会让集群状况更加恶化。

**3. 提前预测、提前行动**

1) 预测磁盘故障，容错单磁盘错误。

**场景复现：**

某厂商的SSD盘某批次存在问题，集群上线运行一段时间后，局部集中出现数量较多的坏盘，但并非所有的盘都损坏。当时并未有单磁盘容错机制，一块磁 盘坏掉，整个机器就被置成不可用状态，这样导致拥有这批坏盘的机器都不可用，集群在一段时间内都处于副本修复状态，吞吐受到较大影响。

**改进措施:**

a) 对硬盘进行健康性预测，自动迁移大概率即将成为坏盘的数据副本

近年来，对磁盘健康状态进行提前预测的技术越来越成熟，技术上已可以预判磁盘健康程度并在磁盘拥有大概率坏掉前，自动迁移数据到其他磁盘，减少磁盘坏掉对系统稳定性的影响。

b) 对单硬盘错误进行容错处理

存储节点支持对坏盘的异常处理。单盘挂掉时，自动迁移/修复单盘的原有数据到其他盘，而不是进程整体宕掉，因为一旦整体宕掉，其他盘的数据也会被分 布式存储系统当做缺失副本，存储资源紧张的集群经历一次这样的宕机事件会造成长时间的副本修复过程。在现有的分布式存储系统中, 也有类似淘宝TFS那样，每个磁盘启动一个进程进行管理，整机挂载多少个盘就启动多少个进程。

2) 根据现有存储分布，预测均衡性发展，提前进行负载均衡操作。

这类的策略设计越来越常见。由于分布式存储集群挂机后的修复策略使得集群某些机器总有几率成为热点机器，我们可以对此类的机器进行热点预测，提前迁移部分数据到相对负载低的机器。

负载均衡策略和副本选择策略一样，需要取舍复杂度和优化程度问题。复杂的均衡策略带来好的集群负载，但也因此引入高复杂度、高bug率问题。如何取舍，仍旧是个困扰分布式存储系统设计者的难题。

## 四 安全模式

安全模式是项目实践过程中产生的防分布式存储系统雪崩大杀器，因此我特别将其单独列为一节介绍。其基本思路是在一定时间内宕机数目超过预期上限则让集群进入安全模式，按照策略配置、情况严重程度，停止修复副本、停止读写，直到停止一切操作（一般策略）。

在没有机器region概念的系统中，安全模式可以起到很好的保护作用。我过去参与的一个项目经历的某次大规模宕机，由于没有安全模式，系统进行正 常的处理副本修复，生生将原本健康的存储节点也打到残废，进而雪崩，整个集群都陷入疯狂副本修复状态。这种状态之后的集群修复过程会因为已发生的副本修复 导致的元信息/实际数据的更改而变的困难重重。 该事件最后结局是数据从冷备数据中恢复了一份，丢失了冷备到故障发生时间的数据。

当然，安全模式并非完美无缺。“一段时间”、“上限”该如何设置、什么时候停副本修复、什么时候停读、什么时候停写、是自己恢复还是人工干预恢复到 正常状态、安全模式力度是否要到region级别，这些问题都需要安全模式考虑，而此类的设计一般都和集群设计的目标用户息息相关。举例，如果是低延迟且 业务敏感用户，可能会选择小规模故障不能影响读写，而高延迟、高吞吐集群就可以接受停读写。

## 五 思考

由于分布式存储系统的复杂性和篇幅所限，本文仅选择有限个典型场景进行了分析和讨论， 真实的分布式存储系统远比这数个案例复杂的多、细节的多。如何平衡集群异常自动化处理和引入的复杂度，如何较好的实现流控和避免影响低延迟用户的响应时 间，如何引导集群进行负载均衡和避免因负载均衡带来的过量集群资源开销，这类问题在真实的分布式存储系统设计中层出不穷。如果设计者是你，你会如何取舍呢?

原文链接:http://www.infoq.com/cn/articles/distributed-storage-system-avalanche-effect

# 黑客内核：编写属于你的第一个Linux内核模块

作者:linux

内核编程常常看起来像是黑魔法，而在亚瑟 C 克拉克的眼中，它八成就是了。Linux内核和它的用户空间是大不相同的：抛开漫不经心，你必须小心翼翼，因为你编程中的一个bug就会影响到整个系统。 浮点运算做起来可不容易，堆栈固定而狭小，而你写的代码总是异步的，因此你需要想想并发会导致什么。而除了所有这一切之外，Linux内核只是一个很大 的、很复杂的C程序，它对每个人开放，任何人都去读它、学习它并改进它，而你也可以是其中之一。

学习内核编程的最简单的方式也许就是写个内核模块：一段可以动态加载进内核的代码。模块所能做的事是有限的——例如，他们不能在类似进程描述符这样 的公共数据结构中增减字段（LCTT译注：可能会破坏整个内核及系统的功能）。但是，在其它方面，他们是成熟的内核级的代码，可以在

需要时随时编译进内核 （这样就可以摒弃所有的限制了）。完全可以在Linux源代码树以外来开发并编译一个模块（这并不奇怪，它称为树外开发），如果你只是想稍微玩玩，而并不 想提交修改以包含到主线内核中去，这样的方式是很方便的。

在本教程中，我们将开发一个简单的内核模块用以创建一个/dev/reverse设备。写入该设备的字符串将以 相反字序的方式读回（“Hello World”读成“World Hello”）。这是一个很受欢迎的程序员面试难题，当你利用自己的能力在内核级别实现这个功能时，可以使你得到一些加分。在开始前，有一句忠告：你的模 块中的一个bug就会导致系统崩溃（虽然可能性不大，但还是有可能的）和数据丢失。在开始前，请确保你已经将重要数据备份，或者，采用一种更好的方式，在 虚拟机中进行试验。

## 尽可能不要用root身份

默认情况下，**/dev/reverse**只有root可以使用，因此你只能使用**sudo**来运行你的测试程序。要解决该限制，可以创建一个包含以下内容的**/lib/udev/rules.d/99-reverse.rules**文件：

```
SUBSYSTEM=="misc", KERNEL=="reverse", MODE="0666"
```

别忘了重新插入模块。让非root用户访问设备节点往往不是一个好主意，但是在开发其间却是十分有用的。这并不是说以root身份运行二进制测试文件也不是个好主意。

## 模块的构造

由于大多数的Linux内核模块是用C写的（除了底层的特定于体系结构的部分），所以推荐你将你的模块以单一文件形式保存（例如，reverse.c）。 我们已经把完整的源代码放在GitHub上——这里我们将看其中的一些片段。开始时，我们先要包含一些常见的文件头，并用预定义的宏来描述模块：

```c
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Valentine Sinitsyn <valentine.sinitsyn@gmail.com>");
MODULE_DESCRIPTION("In-kernel phrase reverser");
```

这里一切都直接明了，除了MODULE_LICENSE()：它不仅仅是一个标记。内核坚定地支持GPL兼容代码，因此如果你把许可证设置为其它非GPL兼容的（如，“Proprietary”[专利]），某些特定的内核功能将在你的模块中不可用。

## 什么时候不该写内核模块

内核编程很有趣，但是在现实项目中写（尤其是调试）内核代码要求特定的技巧。通常来讲，在没有其它方式可以解决你的问题时，你才应该在内核级别解决它。以下情形中，可能你在用户空间中解决它更好：

- 你要开发一个USB驱动 —— 请查看libusb。
- 你要开发一个文件系统 —— 试试FUSE。
- 你在扩展Netfilter —— 那么libnetfilter_queue对你有所帮助。

通常，内核里面代码的性能会更好，但是对于许多项目而言，这点性能丢失并不严重。

由于内核编程总是异步的，没有一个main()函数来让Linux顺序执行你的模块。取而代之的是，你要为各种事件提供回调函数，像这个：

```c
static int __init reverse_init(void)
{
    printk(KERN_INFO "reverse device has been registered\n");
    return 0;
}

static void __exit reverse_exit(void)
{
    printk(KERN_INFO "reverse device has been unregistered\n");
}

module_init(reverse_init);
module_exit(reverse_exit);
```

这里，我们定义的函数被称为模块的插入和删除。只有第一个的插入函数是必要的。目前，它们只是打印消息到内核环缓冲区（可以在用户空间通过dmesg命令访问）；KERN_INFO是日志级别（注意，没有逗号）。

init和 exit是属性 —— 联结到函数（或者变量）的元数据片。属性在用户空间的C代码中是很罕见的，但是内核中却很普遍。所有标记为__init的，会在初始化后释放内存以供重用（还记得那条过去内核的那条“Freeing unused kernel memory…[释放未使用的内核内存……]”信息吗？）。__exit表明，当代码被静态构建进内核时，该函数可以安全地优化了，不需要清理收尾。最后，module_init()和module_exit()这两个宏将reverse_init()和reverse_exit()函数设置成为我们模块的生命周期回调函数。实际的函数名称并不重要，你可以称它们为init()和exit()，或者start()和stop()，你想叫什么就叫什么吧。他们都是静态声明，你在外部模块是看不到的。事实上，内核中的任何函数都是不可见的，除非明确地被导出。然而，在内核程序员中，给你的函数加上模块名前缀是约定俗成的。

这些都是些基本概念 - 让我们来做更多有趣的事情吧。模块可以接收参数，就像这样：

```
# modprobe foo bar=1
```

modinfo命令显示了模块接受的所有参数，而这些也可以在/sys/module//parameters下作为文件使用。我们的模块需要一个缓冲区来存储参数 —— 让我们把这大小设置为用户可配置。在MODULE_DESCRIPTION()下添加如下三行：

```
1. static unsigned long buffer_size = 8192;
2. module_param(buffer_size, ulong, (S_IRUSR | S_IRGRP | S_IROTH));
3. MODULE_PARM_DESC(buffer_size, "Internal buffer size");
```

这儿，我们定义了一个变量来存储该值，封装成一个参数，并通过sysfs来让所有人可读。这个参数的描述（最后一行）出现在modinfo的输出中。

由于用户可以直接设置buffer_size，我们需要在reverse_init()来清除无效取值。你总该检查来自内核之外的数据 —— 如果你不这么做，你就是将自己置身于内核异常或安全漏洞之中。

```
1. static int __init reverse_init()
2. {
3.     if (!buffer_size)
4.         return -1;
5.     printk(KERN_INFO
6.         "reverse device has been registered, buffer size is %lu
   bytes\n",
7.         buffer_size);
8.     return 0;
9. }
```

来自模块初始化函数的非0返回值意味着模块执行失败。

原文链接: http://linux.cn/article-3251-1.html

# 什么是Docker?

作者:oilbeater

尽管之前久闻Docker的大名了，但是天资愚钝，对其到底是个啥东西一直摸不清，最近花了一段时间整理了一下，算是整理出一点头绪来。

官网的介绍是这样的：

Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications....

其实看完这句话还是不明白究竟是啥的，下面就慢慢解释。不过长话短说的话，把他想象成一个用了一种新颖方式实现的超轻量虚拟机，在大概效果上也是正 确的。当然在实现的原理和应用上还是和VM有巨大差别的，并且专业的叫法是应用容器（Application Container）。

## 为啥要用容器?

那么应用容器长什么样子呢，一个做好的应用容器长得就好像一个装好了一组特定应用的虚拟机一样。比如我现在想用Mysql那我就找个装好Mysql的容器，运行起来，那么我就可以使用 Mysql 了。

那么我直接装个 Mysql 不就好了，何必还需要这个容器这么诡异的概念？话是这么说，可是你要真装Mysql的话可能要再装一堆依赖库，根据你的操作系统平台和版本进行设置，有时 候还要从源代码编译报出一堆莫名其妙的错误，可不是这么好装。而且万一你机器挂了，所有的东西都要重新来，可能还要把配置在重新弄一遍。但是有了容器，你 就相当于有了一个可以运行起来的虚拟机，只要你能运行容器，Mysql 的配置就全省了。而且一旦你想换台机器，直接把这个容器端起来，再放到另一个机器就好了。硬件，操作系统，运行环境什么的都不需要考虑了。

在公司中的一个很大的用途就是可以保证线下的开发环境、测试环境和线上的生产环境一致。当年在 Baidu 经常碰到这样的事情，开发把东西做好了给测试去测，一般会给一坨代码和一个介绍上线步骤的上线单。结果代码在测试机跑不起来，开发就跑来跑去看问题，一会 儿啊这个配置文件忘了提交了，一会儿啊这个上线命令写错了。找到了一个 bug 提上去，开发一看，啊我怎么又忘了把这个命令写在上线单上了。类似的事情在上线的时候还会发生，变成啊你这个软件的版本和我机器上的不一样……在 Amazon 的时候，由于一个开发直接担任上述三个职位，而且有一套自动化部署的机制所以问题会少一点，但是上线的时候大家还是胆战心惊。

若果利用容器的话，那么开发直接在容器里开发，提测的时候把整个容器给测试，测好了把改动改在容器里再上线就好了。通过容器，整个开发、测试和生产环境可以保持高度的一致。

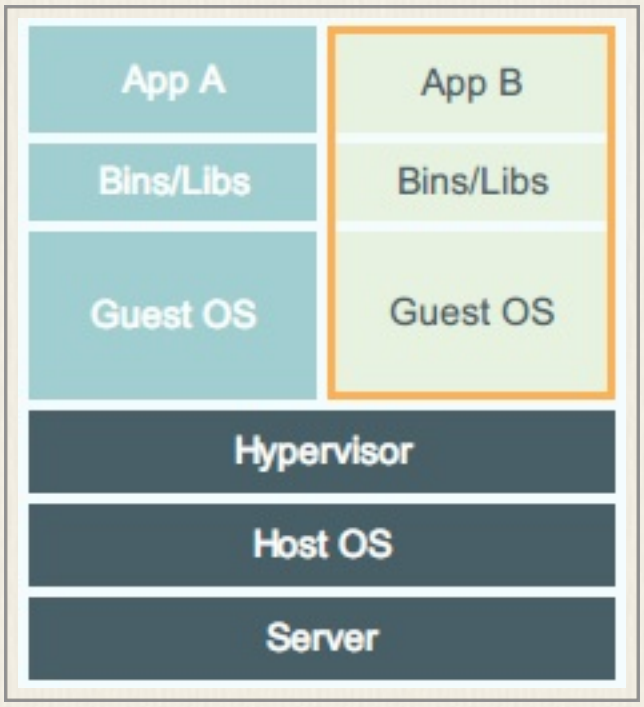
此外容器也和VM一样具有着一定的隔离性，各个容器之间的数据和内存空间相互隔离，可以保证一定的安全性。

## 那为啥不用VM?

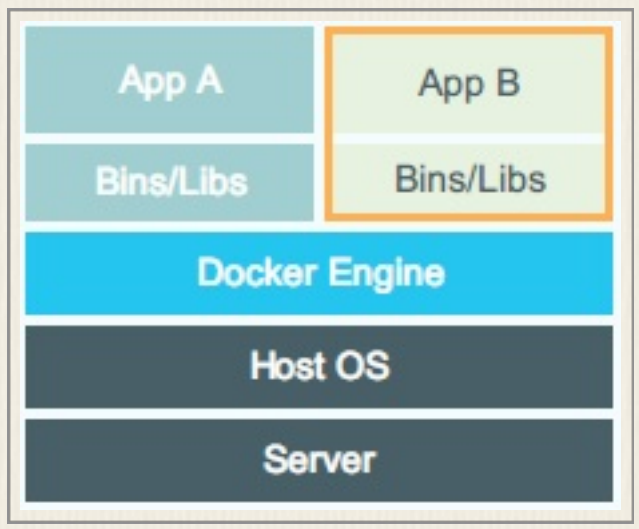
那么既然容器和 VM 这么类似为啥不直接用 VM 还要整出个容器这么个概念来呢？Docker 容器相对于 VM 有以下几个优点：

- 启动速度快，容器通常在一秒内可以启动，而 VM 通常要更久
- 资源利用率高，一台普通 PC 可以跑上千个容器，你跑上千个 VM 试试
- 性能开销小， VM 通常需要额外的 CPU 和内存来完成 OS 的功能，这一部分占据了额外的资源

为啥相似的功能在性能上会有如此巨大的差距呢，其实这和他们的设计的理念是相关的。 VM 的设计图如下：




VM 的 Hypervisor 需要实现对硬件的虚拟化，并且还要搭载自己的操作系统，自然在启动速度和资源利用率以及性能上有比较大的开销。而 Docker 的设计图是这样的：




Docker 几乎就没有什么虚拟化的东西，并且直接复用了 Host 主机的 OS，在 Docker Engine 层面实现了调度和隔离重量一下子就降低了好几个档次。 Docker 的容器利用了 LXC，管理利用了 namespaces 来做权限的控制和隔离， cgroups 来进行资源的配置，并且还通过 aufs 来进一步提高文件系统的资源利用率。

其中的 aufs 是个很有意思的东西，是 UnionFS 的一种。他的思想和 git 有些类似，可以把对文件系统的改动当成一次 commit 一层层的叠加。这样的话多个容器之间就可以共享他们的文件系统层次，每个容器下面都是共享的文件系统层次，上面再是各自对文件系统改动的层次，这样的话极 大的节省了对存储的需求，并且也能加速容器的启动。

## 下一步

有了前面的这些介绍，应该对 Docker 到底是啥有些了解了吧， Docker 是 用 Go 语言编写的，源代码托管在 github 而且居然只有 1W 行就完成了这些功能。如果想尝试一下的话可以看官方介绍了，应该上手会容易一些了。博主也是新手，如有错误欢迎拍砖指正。

原文链接: http://oilbeater.com/docker/2014/06/29/what-is-docker.html

# 布道师徐立：Docker是标准化IT结构的新方式

作者:周小璐

摘要：徐立认为Docker不是一门新技术，而是一种适应时代需要、标准化IT结构的新方式，一种冲击虚拟化的新玩法，并且符合当下分布式协作自成平台的理念，这一理念已经在Github上得到验证，在Docker上会有更广阔的验证空间。

Docker的英文本意是“搬运工”，在程序员的世界里，Docker搬运的是集装箱（Container），集装箱里装的是任意类型的App，开发者通 过Docker可以将App变成一种标准化的、可移植的、自管理的组件，可以在任何主流系统中开发、调试和运行。最重要的是，它不依赖于任何语言、框架或 系统。

Docker 是用 Go 语言开发实现的，而七牛云存储可以说是全球第一个用 Go 吃螃蟹还吃得很开心的玩家，所以Docker在GitHub上一经开源就引起了七牛的注意。徐立认为Docker不是一门新技术，而是一种适应时代需要、标准化IT结构的新方式，一种冲击虚拟化的新玩法，并且符合当下分布式协作自成平台的理念，这一理念已经在Github上得到验证，在Docker上会有更广阔的验证空间，未来必然大有可为，大有作为，事在人为。

**CSDN**：首先请简单介绍下您自己？您从什么时候开始接触Docker的?

**徐立**：大 家好！我叫徐立，悄无声息默默无闻当了十年码农，如今多半是一个屌丝互联网创业者，在给广大高帅富互联网淘金者做送水服务。我在中学开始接触编程，少年时 代偶尔挣点游戏点卡和零花钱，从以前比较偏向业务层的应用开发做到后来比较系统底层的技术，一直保持对互联网技术日新月异的关注与同步。在中国互联网快速 演进的这十多年间，快速经历

了桌面软件，PC互联网，移动互联网，互联网硬件这样一个快速变革的时代，算是在这个最具中国特色的互联网时代背景下土生土长 众多外表粉嫩内心沧桑程序员中的一员。由于当前工作性质的原因，目前大多数时间在做前沿技术的学习与分享交流，人称布道师，主要是分享交流和 Git/Go/Docker 相关的话题。

接触 Docker 还是要和接触 Go 说起，Docker 和 Go 本身有着密不可分的关系，Docker 是用 Go 语言开发实现的。而我和我们的团队早在 2011 年就开始大规模实践使用 Go 开发分布式存储系统，这比 Google 官方发布 Go 1.0 还要早 1 年，也比 Google 官方将 Google 整个下载服务（和我们做过的分布式存储系统殊途同归）由以前的 C++ 换成 Go 来替代还要早 2 年多。由此可见，七牛云存储可以说是全球第一个用 Go 吃螃蟹还吃得很开心的玩家，这比创造出 Go 语言的 Google 公司还要早得多。基于 Go 的语言特性，我们更看重 Go 在多核和分布式系统级领域的造诣，所以很早就有留意基于 Go 做的系统级项目，Docker 就是基于这样一个缘由走进我们观察视角的开源产物。正如我们看好 Go 必然如日中天必将引爆流行成为最主流的编程语言一样，我们很早就注意到在 Github 上开源不久的 Docker，并见证 Docker 后续一发不可收拾迅速成长为 Go 社区乃至整个 IT 业界引发革命的明星产物。

**CSDN**：七牛从什么时候开始使用Docker？主要应用在哪些方面？

**徐立**：作为一名布道师，深知天赋使命，Docker 在第一次公开面世时，就曾引进到我们公司交流学习，探讨其用武之地。同时，作为一家技术气质特征明显技术追求欲望强烈的这么一家技术驱动型公司，一直在积极探索和前瞻技术结合产生的创新体验。Docker 的典型玩法包括 构建持续集成（CI）/ 持续部署环境（CD）、打包和批量发布程序，以及搭建独立的 PaaS（Platform as an Service）平台等。大多数人选择的应用场景是用于构建 CI/CD ，而我们选择了一条少有人走的路。

众所周知，七牛云存储是一个面向开发者的公有云服务。大部分开发者对公有云存储服务的认知就是 Amazon S3；但现在，七牛云存储比 Amazon S3 有着更为丰富的功能特性。如果说 Amazon S3 开创了云存储的 1.0

时代，那么七牛云存储则开创云存储的 2.0 时代以及 3.0 时代的大门。开发者都知道 Amazone S3 最主要的作用就是提供静态文件的存储和获取，也就是 PUT（上传） 和 GET（下载），而七牛云存储在这个基础之上扩展了更多对开发者尤为便利的支持，按照业务形态，七牛云存储可以这样划分：云存储（Input） -> 云处理（Process） -> 云分发（Output）。云存储就是我们开发者理解的 PUT，包括分布式多节点加速上传；云处理主要指静态文件存放上来之后在云端进行的个性化处理，这些处理包括常见的在线图片压缩裁剪转换水印等处理、Office系列的文档转换处理、语音处理、视频转码流媒体处理、自定义的数据处理等等；云分发就是 CDN 功能，目前七牛全国的CDN节点超过 1000 家，而且平滑扩展到海外。Web或App等应用的开发者，只需要针对服务端做到动静分离，将静态文件放到七牛后，即可大大节省带宽成本和开发精力同时将用户体验做到足够极致。传统开发方式要用至多 18 个月才能上线一款互联网产品的节奏，现今只需至多 1.8 个月即可上线运营，且无任何扩容和是否高可用的心智负担，可谓真的多快好省。

在理解七牛的业务形态后，其次就很容易联想到七牛会在哪方面应用Docker。首先，七牛的分布式存储系统和磁盘关系紧密，不太可能用 Docker 去部署存储系统（这和 Docker 的运行机理有关，具体原因在线直播时我会展开讲解）。其次，关于持续集成测试方面，我们有用到的 drone.io 和travis-ci 或多或少有用到 Docker 或 LXC 技术。但这些都还是属于很外围的应用场景，更好玩的真正用武之地，在于我们现在可以用 Docker 实现各式各样的数据处理，也就是说， 我们现在完全可以用 Docker 把七牛的云处理玩得千变万化，对于开发者来讲，真的实现让天下没有难写的服务端代码。比如说，开发者想要的一般图片处理功能七牛云存储已经提供接口了，但 是需要一个图像识别程序怎么办呢？这个时候如果开发者自己有写过这样的程序完全就可以部署到七牛的云处理引擎里边来，这样子数据在七牛上，数据处理程序在 七牛云处理模块的容器里，都在同局域网，就可以轻而易举地实现数据的就近处理输出开发者预期想要的结果。

**CSDN**：目前企业应用Docker最大的困难是什么？

**徐立**：Docker 的学习成本并不高，我认为最大的困难在于企业观念和思维的转化，是否接受并习惯这一新的理念，最典型的包括传统开发方式和运维方式习惯的迁移。

**CSDN**：据您了解，Docker 目前在国内的发展状况如何？未来会如何发展？

**徐立**：我自己受邀参加过几次公开活动，去各个地方布道宣讲 Docker，现在能够看到的状况是星星之火呈燎原之势。Docker 不是一门新科技，而是适应时代需要，一种标准化 IT 结构的新方式，一种冲击虚拟化的新玩法，以及符合当下分布式协作自成平台的相似理念，这一理念在 Github 上验证成功过，在 Docker 上会有更广阔的验证空间，未来必然大有可为，大有作为，事在人为。

**CSDN**：你在简介中说自己是布道师，这是个什么样的职业?

**徐立**：在 外界听演讲的观众看来，这人看起来是一个大牛，说起来也像个大牛，晃起来还像个大牛，一定就是一个大牛。在客户看来，这真是一个超级实用的万能顾问。在老 板看来，这是一个不错的技术出生懂产品的业务员。在同事看来，这是一个好打配合超给力的魔法师。在另一半看来，这是一个携程在手说走就走的漂泊流浪汉。在布道师自己来看，这是一个行者。

原文链接: http://www.csdn.net/article/2014-06-27/2820423-qiniu-Docker

# 高频交易软硬件是怎么架构的?

作者: 知乎用户

这个链接给出了大量信息。http://adtmag.com/Articles/2011/07/29/Why-HFT-Programmers-Earn-Top-Salaries.aspx?m=1&Page=1

但问题是这里的信息覆盖了所有可能。从FPGA,GPU到C/C++到Java/.NET到Erlang/Haskell。有意思的是有人在回复中说高盛被偷走的主要是Erlang代码。

很显然，不同公司的方案大不相同。但让我费解的是，不同消息来源暗示的关注点完全不同。比如说，有消息来源说只有性能关键的少数部分是C++，而也有消息来源说latency已经达到了nanosecond级别，而且至少都是us级别。

latency 达到nanosecond级别是非常疯狂的，因为航空航天的realtime只能到几十us的程度。这意味着任何操作系统都将成为累赘，甚至baremetal的软件也达不到此要求，唯一可能是把逻辑全部写在FPGA里。无法想象能用FPGA实现极其复杂而且高度变化的数学模型。

好 吧，假设有这样的硬件存在并可商用，那么如此辛苦得来的高效如何与C++甚至Java并存呢？别说软件，哪怕是memory bus都将成为瓶颈。而又有消息指出被广泛使用的操作系统是Linux。Linux最多介入下软实时，这还都是得在mainstream上单独打 patch才能勉强过关，在比硬实时还高一个级别的场景里不可能用的上。

还有些信息来源指出FP的广泛应用，这可以理解为对程序正确性的要求，但在密集浮点计算领域根本没FP什么事，占统治地位的还是FORTRAN。不知道有没有把FP编译成CUDA目标的编译器存在?

诛心之论，是高频交易界故意模糊视线并人为抬高门槛。哪怕高频交易中的经济和数学模型再天外飞仙，在软硬件架构上还是得落地，因为目前技术极限在这摆着，再加上相互矛盾的信息来源也暗示了故意搅浑水的可能性。

看了几位的回答后，意识到有不少概念没讲清楚，连问题本身也没有明确。遂更新如下。

1. 具体在问什么

标题其实很清楚，就是想知道一个实际运行的高频交易系统的软件和硬件架构。其中应该包括网络拓扑，硬件资源，软件结构等等等等。

但我还顺便掺入了我对HFT报道的怀疑，也就是到底HFT采用的硬件和软件是不是跟广告中说的那么神奇。

2. 对latency的定义

按原文中的描述，是"ultra low latency trading"，并且是"wire-to-wire"的。我的理解是，一个HFT的计算节点，有能力在几百ns内处理完一个网络上接收到的信息包，并将结果返回到网络上。

这个信息包从物理上来说是一个IP packet，所谓的“接受”和“返回”，指的是从直接连接这个计算节点的router的角度来观测的。

3. FP指的是Functional Programming。因为那篇文章反复提到了Erlang/OCaml/Haskell/Lisp。我之所以会提到编译器，是因为文章同时反复提到了CUDA，因此我猜测既然又要利用FP逻辑不容易出错的特性，又要利用CUDA的SIMD/SIMT特性，那这个接口工作交给编译器是不是更方便些。

4. realtime

在这里我语焉不详并混淆了概念。确实，软实时还是硬实时指的是任务完成是否有guarantee或者说上限。我真正要表达的意思是，以我的了解，不用硬件解决方案不可能达到ns数量级。（下面会补充说明）

5. 为什么我理解HFT的逻辑是复杂并多变的

因 为首先做HFT的都是理论物理，数学方面的博士教授甚至菲尔兹奖得主。所以其逻辑肯定是非常复杂的。其次，从文章以及其它描述HFT码农的文章来 看，HFT码农面对持续的高压，并且要快速做出修改，甚至是online的修改，所以我猜测软件需要实现的逻辑需要持续的修改。

6. 关于现在工业上的network stack能做到多快，我有一些第一手的资料

我个人提交了两个fea-
ture的补丁给DPDK并被接受，所以我还是比较了解DPDK的。

另外我还propose了一些Intel 10G NIC driver的补丁。

并且我也实际测试过DPDK+10G PCIE的thruput和latency。

结果很不妙。关键不在DPDK或者网卡或者cache或者CPU或者bus，而在Linux。

只要是跑在Linux上的-无论usr space还是kernel module，都注定达不到ns的latency，甚至连us都达不到。这都归咎于无处不在的soft irq和sched。是的，我知道affinity和cpuisol，没用，真的。

虽然我没接触过连PCIe都嫌弃的方案，连DMA都订制的方案，但我认为瓶颈不在那，根据Amdahl那些改动没有用。

另外，这还没考虑HFT自己的逻辑，按那篇文章的介绍，不管是FPGA还是GPU实现的浮点计算，那么光是数据从网卡到bus再到GPU然后再回来，加上GPU计算--这都还没考虑CPU的调度带来的损耗--能在1000个cycle内跑完就算很不错了。

ok， 废话可能太多了点。总之我想说的是，从文章（SO上也有类似问题及回答）来看HFT有一些不差钱的解决方案，但这些不差钱的方案却又不是整个系统的瓶颈所 在，相反，HFT还为了向正确性/可维护性妥协而采用了大量减慢系统的方案。这是怎么一回事？到底是因为HFT就是这样（是的我就是钱多，我吃一碗倒一 碗），还是HFT的宣传全是不符合事实的?

作者:董可人

有几个误区是要先澄清的。

一，HFT不一定是套利算法。事实上HFT做的最多的业务是做市，可以是把商品从一个交易所倒卖到另一个交易 所，也可以是在同一个交易所内部提供某种商品的流动性。这两种方式的共同点都是让人们可以特定地点买到本来买不到的商品，所以本身就是有价值的，收服务费 就可以盈利。反而是套利算法是投入高风险高的生意，一旦市场环境变化就要重新研发，不是长久的生意。

二，延迟和带宽是不同的概念。低延迟不等于高数据量，事实上大部分时间交易数据流量并不大，一个market一天最多也就几个GB。但HFT系统需要在流量高峰时也能快速响应，所以更看重延迟。

三， 一个HFT业务包括从主机到交易所的整条通信线路，在这条线路上有很多段不同的延时，是需要分开讨论的。如果是做跨交易所的交易，首先需要考虑的是两个交 易所之间的网络延迟。当数据通过网络到达主机的时候，有一个最基本的tick-to-trade延时，是指主机接收到数据到做出响应所需的时间。但这个东 西的测量很有技术含量，根据不同的测量方式，它可能包括或不包括网卡及网络栈的处理时间。所以拿到一个HFT系统的延迟数据时，首先要搞清楚它指的是什 么，然后再来讨论。

接下来说说做为一名从业者，我对各个层面的理解。

首先网络架设上光纤肯定是最差的方案。国外 几个主要的交易所之间基本上都有微波（microwave/milliwave）线路，比光纤的延迟要低一个数量级，延迟敏感的应用一定要选择这种线路。 但微波技术有两个主要的缺点，第一是在空气里传播受天气影响很大，刮风下雨都会导致通信受损，有时直接故障，所以一定需要有备用的光纤线路；第二是带宽太 小，如果是跨交易所的业务，不可能通过微波来转移市场数据，只能用来收发下单指令。第二点给网络服务商提供了一定操作的空间，比如可以自己做一点数据压缩 和抽样，就可以在微波线路上提供一个微缩版的市场数据，非常有价值。这块网络服务本身就是一个独立的业务了，一般所说的colocation也是由服务商 负责的，HFT主要需要的是选择适合自己的服务商。

网络线路确定以后，数据就送到了HFT主机。这时候需要决定网卡的方案，专用的网卡除 了自身硬件的设计外，一定需要的是切换掉系统自带的kernel space TCP/IP stack，避免昂贵的context switching。这个层面上FPGA是很有应用价值的，因为可以做一些额外的逻辑处理，进一步解放CPU。

网络部分的问题解决以后， 最后就是核心的业务逻辑的处理。这部分也许会用到一些数学建模，但是没有什么神话，不是什么菲尔兹奖得主才能搞的东西（那些人的用武之地更多是去投行那边 做衍生品，那才是真正需要高等数学的东西）。很多时候核心的还是延时，这个在计算机内部分两个部分，一是core的使用率，比如irq balance，cpuisol，affinity等，主要是要尽可能的独占core；另一个是cache invalidation，从L1/L2/L3 cache到TLB，memory layout之类都要仔细考虑，这个更多考验的是对体系结构的理解和程序设计的功力，跟语言是C++还是erlang的关系不大。具体选择那种语言，主要 是取决于公司的技术积累和市场上的技术人员供给。

对于FPGA，我同意Nil的回答，业务逻辑烧到硬件里的开发，调试成本和周期都是很难承受的，不看好做为长期发展的路线，这个东西其实和套利，数学模型一样是赚外行眼球的东西。但做专用的网络设备却是比较有优势的。

操作系统同样是一个不需要神话的东西，普通的linux已经有足够的空间用来做性能优化。简单说，一个企业级的linux（如redhat）加上通用的架构（intel主流处理器）足以做到市面上已知的最低延迟，不必幻想有什么奇妙的软硬件可以做到超出想像的事情。

最后需要提醒大家注意的是，其实做一个低延迟系统，首先需要考虑的不一定是延迟能降到多低，而是怎么测量系统的延迟？对一个HFT系统来说，所谓的 tick-to-trade延迟，一定要有既精确又不影响系统性能的测试方法才有意义。可以想像一下，最理想的测试场景一定是你的系统真正运行在直连交易 所，有真实的市场数据传入的情况下，并且测试的代码就是真正的交易算法时，得到的数据才有意义。如何得到这个苛刻的测试环境，以及如何测量系统的各个部分 的延迟，是一个非常有技术含量的工程，难度往往并不亚于系统设计本身。

原文链接: http://www.zhihu.com/question/21922144

# 码农故事：一位中级程序员的自白

译者:王伯

我是一名中级程序员。

我有相当不错的基本技能。我犯了足够多的错误才明白为什么那些被称为错误。我很清楚我还需要了解更多东西。最重要的是，我知道那些东西大概是什么，并且我正在努力而积极地提升自己。

勇敢地承认自己不过是水平一般的程序员，这花了我一些时间。我不再感觉有必要去抓住那些我并不了解的观点。当人们发现我对某样东西不了解时，我也不再感到害怕。

事情并非从来如此。你可能对此不以为然，但是我曾经自诩为编程大师。

这种对自己能力的不正确的评估，很大程度归因于我在一个相对封闭的环境中学习技能。在过去那些日子里，有电脑就已经很特别了；更不用说知道如何使用了。

在我自己看来，我当时是一个知识渊博并且经验丰富的程序员。在我不到20岁的时候我已经用C++、Pascal、C#、JavaScript写过程序。当然我最引以为傲的是，曾经徒手用PHP编了一个电子商务平台。

事实上，我可能只是人们平时谈话中提到的“我有个朋友的儿子很会写网站”。我和别的程序员没有任何交流，所以我仅有的比较对象是我周围的人；要么是 一些根本不在意电脑的人，要么是那些会用电脑，但是在IE窗口中塞了5个没用的工具栏的人。那些可能会说“我的网坏了”这种话的人。

接下来这个故事就是讲我如何产生自己很厉害的幻觉的。

## 天才的起源

当我九岁的时候，我的一个朋友家里有卫星电视。而在我们家里，我们只能收到四个英国的频道（你能想象第五频道出现之前的日子吗？），我热切地盼望有 一台普通的电视机。我们所需要的只是那些“卫星盘子”，或者我称为“卫星”的东西——那样我就随时可以看QVC台或者Eurosport台。由于隐约意识 到自己的某种天分，我开始搭建自己的卫星！我的设计包括了一把打开的伞和一条铜质音频线，一段接在伞的金属柄上，另一端接在电视机天线上。必须承认我的设 计有一些缺陷，并直接导致我没有得到想要的结果。但是这个小故事仅仅想表达我童年和青少年时期对技术的渴望。我认识的人中从没人想过制造“卫星”。

几年后，当我父亲的办公室得到一个14.4k的猫时，我成为了最早一批网民一员。我能回忆起花了整个星期六下午的时间等待这个火焰漫画图标被加载，每个帧的动画大概要过一分钟才显示。我甚至用Netscape搭建了我自己的网站。由于不知道互联网的架构，我把所有的HTML文件存放在本地，并且期待有一天他们会出现在互联网上。然而这些细节并没有削弱一个事实：我认识的人中没有一个制作了他们自己的网站。

在我十多岁的时候，我发现了自己天才中的黑暗面。在装备了Jolly Rogers的食谱后， 我和一群小伙伴们准备动摇整个九十年代英格兰的技术和道德根基。破解电话系统是我们的专长。我们用手提式声音耦合器和公用电话，给我们在ICQ上认识的美 国姑娘们打免费国际电话，以及在私人交换机上设立语音信箱。最终学业和滑板阻止了我们在这条路上越走越远，如果没有这些干扰，我们无疑已经在制造凝固汽 油，黑进政府网站并且徒手杀人了。尽管我们没有把自己的能力发挥到极致，但事实是除了我们没有其他人哪怕拥有声音耦合器。

尽管到那个时候我已经经历了一些冒险和失败，我还是缺少一些东西。我的想法总是要超前我自身能力好几步 – 正如在“卫星”一节里体现出来的。我需要一种把我脑海中想法表达出来的方式。我需要一个直接的介于我想象和现实之间的接口。

## Fuck 生成器

真正的转机出现在我十四岁的时候。我购买了一份PC Plus杂志，其中附赠了带有完整版Borland C++编译器的CD。我安装了，并且认真学习了杂志上的“hello world”教程。

就这样，一个崭新的世界在我面前打开了。物质世界对于我想象力的限制消失了。我的创造力被解放了，我脑海中的大教堂要成为现实了！我该把这个新工具用于怎样崇高的事业呢？很显然，Fuck生成器。

简单而优雅的Fuck生成器是一个命令行程序，也是我即”hello world”之后第一个里程碑。程序开始运行后会提示用户输入一个数字n，然后它会输出字符串”fuck”，n次。最后用户被提示可以重复以上过程或是退 出。尽管功能有限，我还是沉醉于我所品尝到的成就。这是任何程序员都能享受到的一种快感，即看着机器执行你的命令，不管这个任务有多简单。它在运行了，并 且你知道为什么它能够运行。它除了在那里运行不会做任何别的事。

过了些时日，另一期的PC Plus附赠了一个完整版的Borland Delphi。有了这个，我把程序升级为带有窗口界面并且可以随机生成彩色的4种不同的脏话。当别的孩子在玩PlayStation的时候，我正在投身于 一项更有意义和创造性的事业，我在生成很多fuck。

到那时，一切都预示着我是注定要成大事的。我要向世人展示我真正可以做的事情。

## 我的巨著

在90年代晚期，我为一家小型并且扩张迅速的邮件订购零售商创建了一个网站。一开始，这个站点只包含一些静态的页面——关于商品的小册子，一个导航菜单和一个访问数量计数器。

当我们的访问量越来越大时，我们决定加入电子商务功能。我们遍历了一些现成的工具包，它们的质量从差到极差不等。我印象中第一个版本大部分建立在摆 弄cgi脚本以及怪异地把<select>元素用于几乎所有的用户交互部分之上。之后的一个版本是充斥着framesets和 Javascript的怪物——远

在Javascript成为举世皆准的构建应用功能的方式之前。另一个版本是由微软的Access数据库驱动的。

不久后我们意识到，如果我们想要一个真正可用的甚至体面的在线商店，我们需要一个自定义解决方案。我想到了我过去的成功经验：fuck生成器系列， 以及截至那时我所编写的优秀网站，这其中：我的 Manic Street Preachers吉他谱收藏网站非常具有权威性。我认为是时候看看我能真正做些什么的时候了。我要自己从头开始干。

从头开始？即使那个时候开源框架已经存在，我也不会知道他们。我有自己的计划。我买了一本关于PHP和MySQL的书，一边学习一边着手搭建新的网站。

幸运的是，这本书把一个非常简单的购物网站作为它的核心例子。所有的部分都在那儿：“category.php”会列出一个目录中的所有物品；“product.php”会显示商品信息以及把该商品加入购物车的按钮；以及最重要的“cart.php”，它是所有奇迹发生的所在。这就是我想要 的东西！

我孜孜不倦地学习这个例子，充满自信地实现所有巧妙的而且毫无疑问也是最新潮的技术 – 那些方便的“mysql_”函数；用于建立查询的字符串连接函 数；把不同的函数放进“functions.php”文件；通过加入“header.php”和“footer.php”来维护整个网站的一致性；为了代 码的快速运行而回避了笨重的面向对象的设计方式（管它是什么玩意）。我的技能在飞速成长。

像一个人的王国一样，我建造了高塔和迷宫般的地道。我每添加一个特性，就好像整个结构在向天空伸展同时也向地下蔓延。顾客帐户、商品评价、购买历 史、优惠点数、帐单号、特殊优惠、日志、 A/B测试、支付信息加密，等等。一个蔓延的迷宫，一整个星系的函数，大的小的，缓缓围绕一个不变的核心：“cart.php”。

经过八个月的激情工作，我终于完成了。

现在，你们这些读者一定在期待我会详述当网站正式运行时发生了怎样恐怖的事情。恐怕我要让你们失望了。

它成功运行了。

## 最糟的方法

尽管我现在把这当作我最糟的设计，但是这个东西确确实是能够运行。它在每一个糟糕的教程，每一个反php的帖子里都能找到。搅成一团的代码？是的。不一致的数据和方法名称？是的。介绍和业务逻辑混在一起？是的。魔幻数和全局变量？是的。

对我而言，面向对象的设计只是一堆不必要的开销和公式化的代码，并且有很多片面的理论支持我的观点。我知道有关测试的所有，点击一些你设计的特性，看上去不错，上传运行。我不太知道别的架构，但是据我所知，我所采用的是最明智的方法。

一些事实能“证明”我所做的都是正确的：我从零开始，白手起家，用智慧创造了一个功能齐全的电子商务站点。更重要的，它运行完好并且还在扩张。

在我的眼里，我和那些写了亚马逊的程序员们没什么太大区别。当然亚马逊要大一些，但是我没有看到任何我的网站不能扩张成那样的理由 – – 尤其考虑到我采用的高速运行的架构。

我认为我的技术水平已经到了巅峰了。并不是说我对学习新技术不感兴趣了，我只是不再对此感到紧迫。毕竟我创造了一些不错的产品。任何在此之上的东西只是附加奖励，是蛋糕顶端的樱桃而已。

## 回到地表

我很遗憾，我在这种心态下生活了好几年。我只是将一小部分时间用在这个网站上，而把主要时间用在完全不同的领域。在之后多年的维护和偶尔添加特性的 过程中，我确实意识到了之前做的一些选择是有问题的。我意识到有时候要花很长时间才能找到我要找的文件。有时候当我做一个改动时，一些看上去毫无关联的地 方会出现bug。

我的学习没有停止，但它确实进展缓慢。我意识到我曾经写的mysql函数是有风险的，因为后面版本的PHP减少了对它们的支持。在一段时间里，我克 服对此的恐惧的方法是坚信我的无懈可击的设计可以弥补这些风险。毕竟我尝试了所有形式的我能找到的SQL注入，一切看起来都没有问题。

去年的一天我接到了一个紧急电话，网站挂了。所有的请求都得到500错误。在工程师们重新启动并且分析了事故原因后，这被证实是一起来自国外的sql注入攻击，是我从来没见过的一种。

好吧，我想，这也许是我该转向PDO的时候了。

## 觉悟

当我坐下来准备重写所有的数据存取方法时，我意识到了一些深层次的问题。我意识到这将会很困难。而且我知道为什么它会这么困难。

因为这些方法散落在所有地方；因为我无法知道是否会不经意地破坏一些东西；因为代码是如此不一致以至于我要小心地研究不同对象的细微差别；因为很多 代码和别的部分紧密相连，这也会导致我会不小心造成破坏。简单地说，这将会很困难。不仅因为所有这些坏的实现方法，还因为我对它们所将造成的后果缺乏预 见。

所有的辩护，借口，逃避都无法继续下去了。我错了。我不是那个幻想中的天赋卓越的程序员。这么多年来，我一直都没有认清这一点。

我的愚蠢已经显而易见，尽管这对我的自尊心是极大的打击，但这也是很宝贵的一个教训。我通过亲身经历（而且是非常痛苦的），学到了为什么做一件事的 方法有对错之分。这不仅仅关系到品味或者时尚。这不是比谁的方法更聪明。正确的方法可以在现实生活中找到，并且能让你和那些使用你代码的人的生活更好。错 误的方法让人沮丧，浪费时间。我在这里不想说明哪些东西是组成“正确方法”的要素。只要说不是我做的那些就够了。

## 真正的错误

我实现了PDO。同时开始第一次使用PHPUnit。我决不想尝试通过单元测试去改造那样的代码。

现在我有意识地迫使自己无论何时都尽量去学习。我正在读一些每个程序员都应该读的书。我在关注别人的博客。我在收听播客。我会看会议视

频。我正在参加一些当地的社团并且在其中做演讲。我在做副业并且挑战自己学习新的技术。我在学习用正确的方法做事。

对你们所有献身于这项事业中的人来说，有一个对我们很重要的有利条件。即编程是这样一个完全抽象的活动，任何其他领域都会受到的现实世界中的限制在这里不存在。在这里，你的极限是你自己。

我要以一些真正的箴言结束这个故事。我在开始写这片博客的时候正好刚看完第二版的《代码大全》。在书的最后，第825页的底部，作者准确地描绘了我在写这篇文章时的想法。可以说他只用了两句话就表达我在这数千字里想表达的东西：

**“作为一个初学者或者进阶者，这并没有什么错。当一个有能力的程序员而不是领导者，这也没有什么错。真正的错误是，当你知道应该如何去提高时仍然选择做一名初学者。”**

译文链接: http://blog.jobbole.com/72179/

原文链接:
http://www.michaelbromley.co.uk/blog/65/confessions-of-an-intermediate-programmer